TURING 图灵原创

■ 季昕华、徐羽作序，连城/胡熠/武泽胜/肖磊/靳志辉联袂推荐
■ 腾讯专家首次分享Spark最佳实践
■ 基于真实数据，用案例分析全面解读大数据应用设计

Spark最佳实践

陈欢 林世飞◎著

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明



陈欢

腾讯资深程序员，15年编码经验，曾任职网络安全、互联网金融等部门，亲手从零建设了财付通业务的Spark集群，并使之同时支持SQL、实时计算、机器学习等多种数据计算场景。他目前就职于腾讯社交与效果广告部，从事大数据分析工作。



林世飞

腾讯资深研究员，2005年加入腾讯，先后在无线产品、安全中心、搜索平台、开放平台、社交与效果广告部等部门从事开发和团队管理工作。他对网络安全、搜索引擎、数据挖掘、机器学习有一定了解，热衷知识传播和分享，曾获腾讯学院2009年年度优秀讲师。目前，他就职于社交与效果广告部，负责广告系统相关的研发工作。

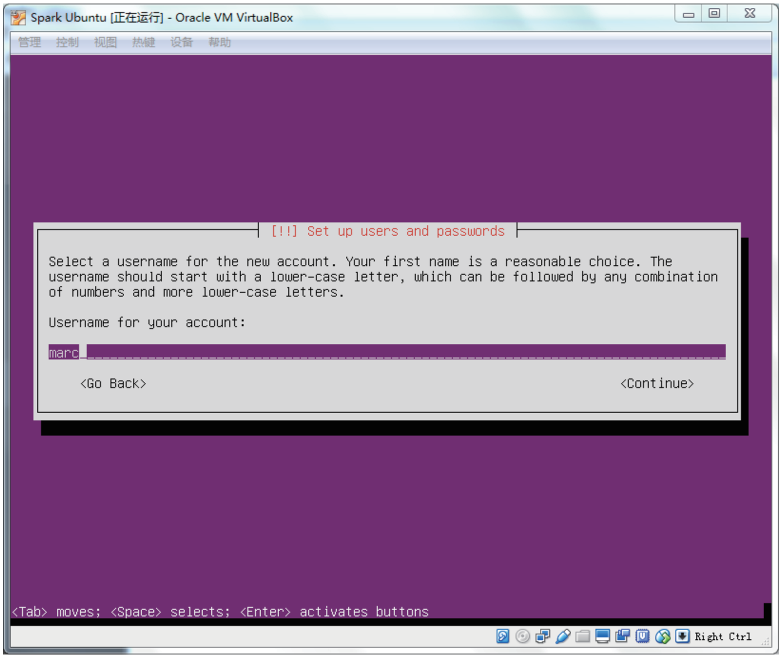


图 2-9　添加账号

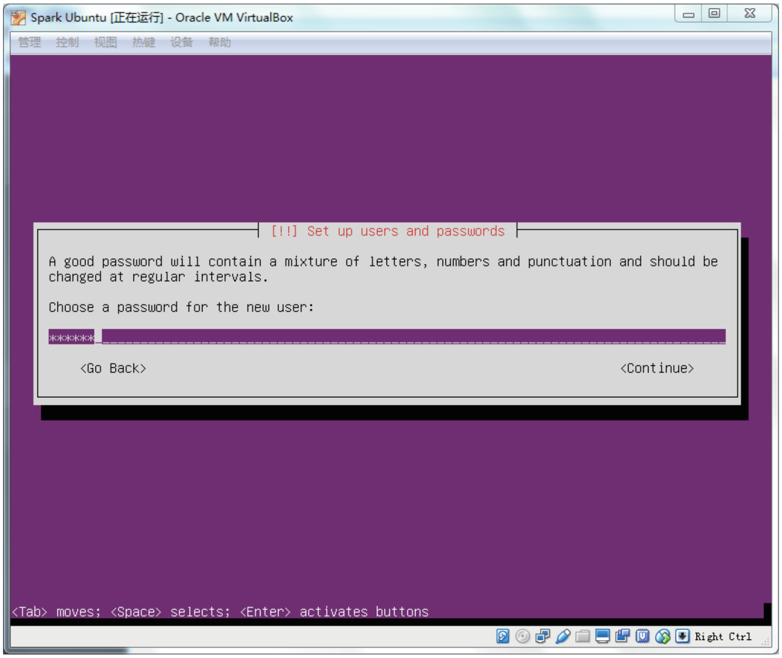


图 2-10　设置密码

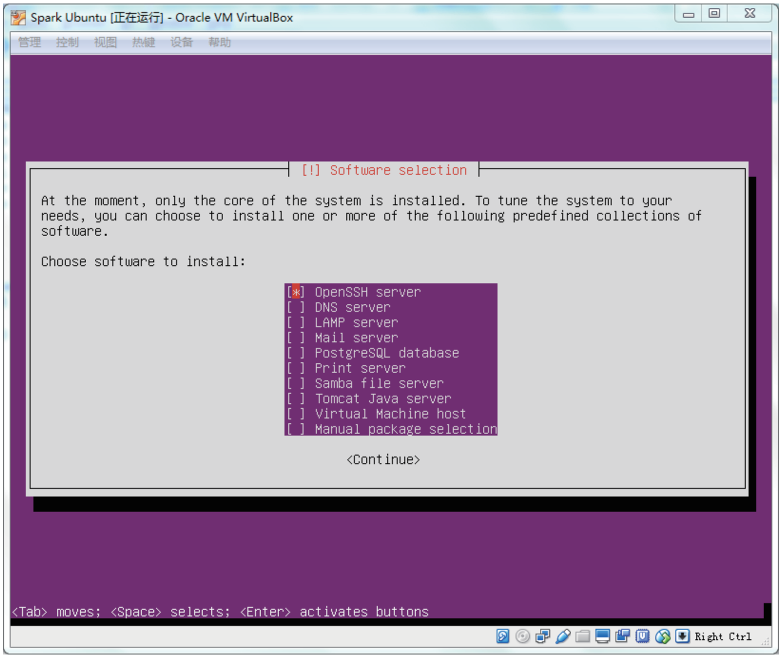


图 2-11 选择 OpenSSH server

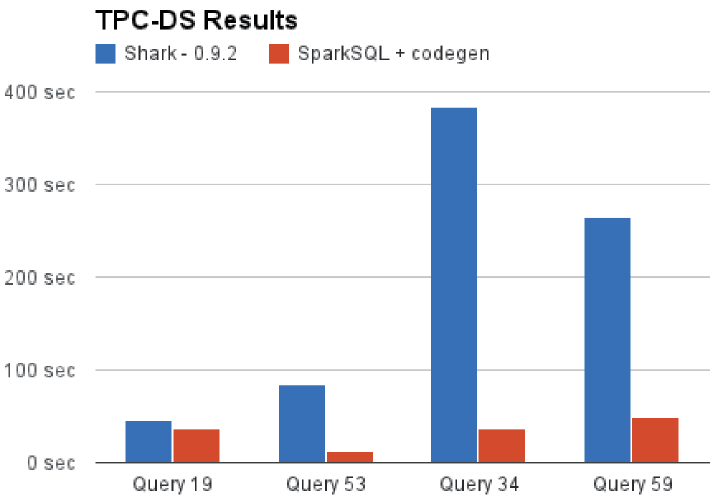


图 5-4 对比性能测试结果

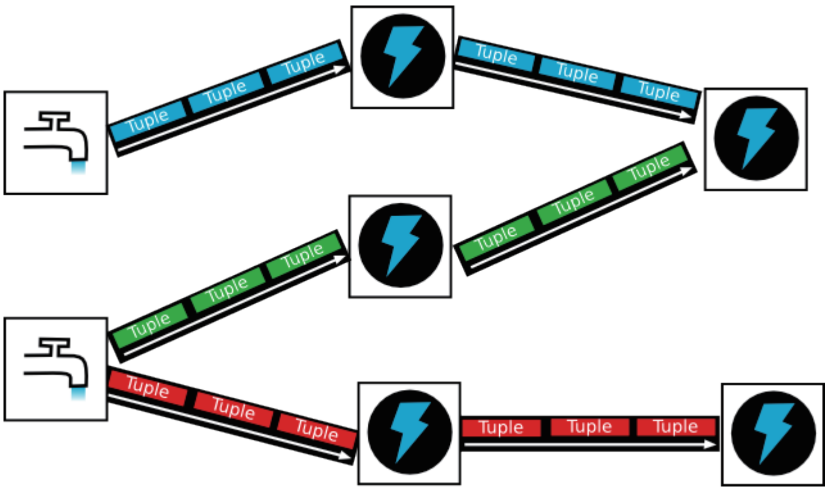


图 6-10　Storm 结构示意图

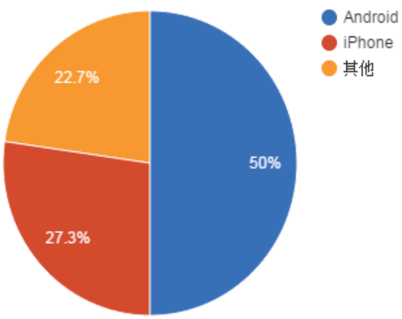


图 6-11　终端类型分布图示例

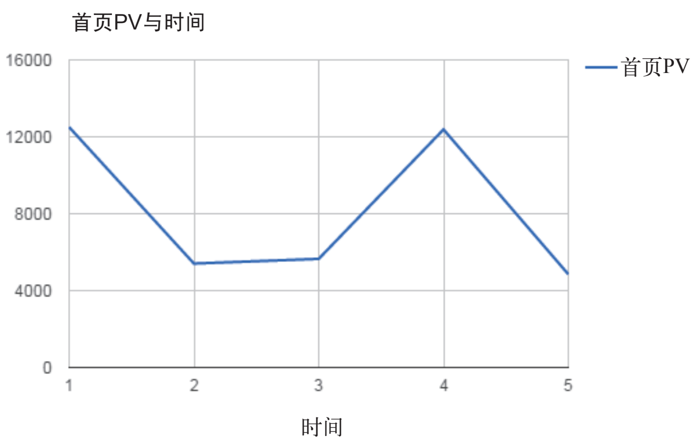


图 6-12　首页 PV 趋势图示例

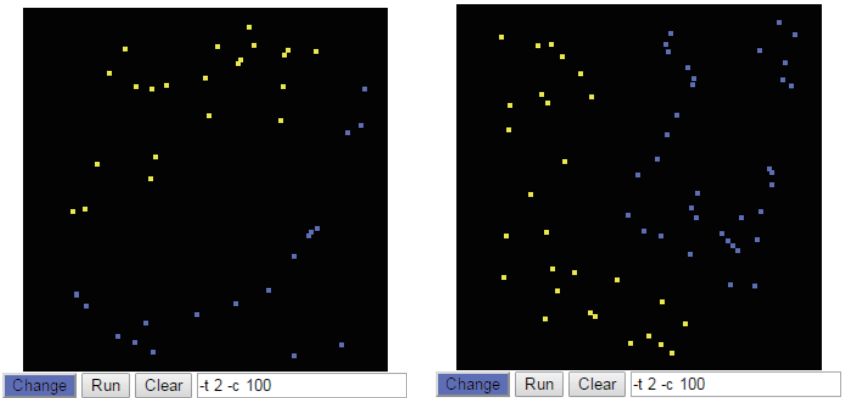


图 8-1　SVM 算法效果演示：学习的对象

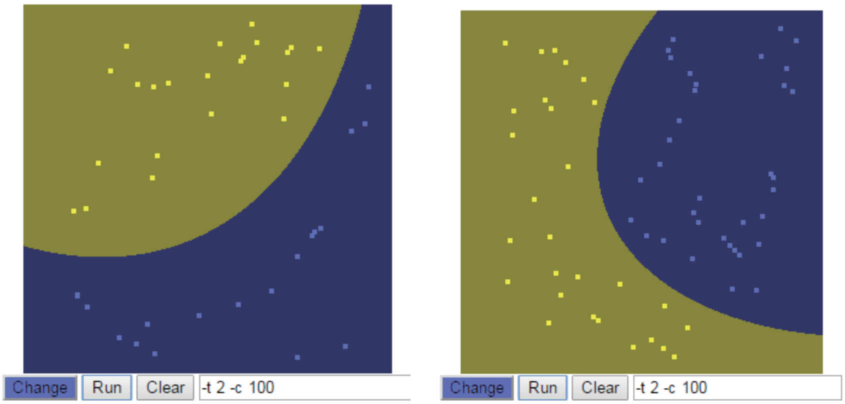


图 8-2　SVM 算法效果演示：学习的效果

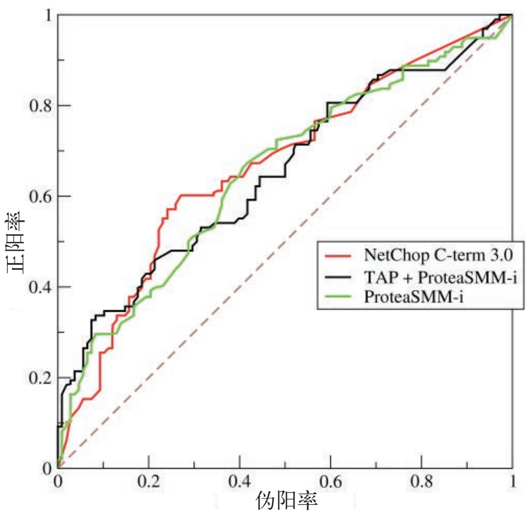


图 8-9　ROC 曲线示例（来自维基百科）

TURING 图灵原创

Spark最佳实践

陈欢 林世飞◎著

人民邮电出版社
北京

图书在版编目（CIP）数据

Spark最佳实践 / 陈欢，林世飞著. -- 北京 : 人民邮电出版社，2016.5
（图灵原创）
ISBN 978-7-115-42228-6

Ⅰ. ①S… Ⅱ. ①陈… ②林… Ⅲ. ①数据处理软件 Ⅳ. ①TP274

中国版本图书馆CIP数据核字(2016)第083128号

内 容 提 要

本书是 Spark 实战指南，全书共分 8 章。前 4 章介绍 Spark 的部署、工作机制和内核，后 4 章分别通过实战项目介绍 Spark SQL、Spark Streaming、Spark GraphX 和 Spark MLlib 功能模块。此外，本书详细介绍了常见的实战问题，比如大数据环境下的配置设置、程序调优等。本书附带的一键安装脚本，更能为初学者提供很大帮助。

本书适合大数据开发、运维等相关从业人员学习参考。

◆ 著　　　　陈 欢　林世飞
责任编辑　毛倩倩
责任印制　彭志环
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　印刷
◆ 开本：800×1000　1/16
印张：14　　　　　　彩插：2
字数：339千字　　　　2016年 5 月第 1 版
印数：1 – 4 000册　　　2016年 5 月北京第 1 次印刷

定价：49.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第 8052 号

序　一

随着互联网的快速发展，特别是云计算的普及，大数据日益受到人们的重视，最近几年全球数据量以每年约 50%的速度递增。大数据正在事实上改变着我们的思维和生产方式，未来人类社会的精神和物质世界都将构建在数据之上。2015 年 9 月，国务院印发《促进大数据发展行动纲要》，确定了大数据发展的国家顶层设计，大数据与各行各业的结合已是行业未来发展的必然趋势。大数据在中国已经全面生根发芽，大数据创业企业也迎来了一波新的发展机会。

学习大数据技术、使用大数据技术为各行各业服务是每个 IT 人都梦寐以求的事情。在中国，BAT 的大数据员工是最幸运的一群人，因为他们手上拥有最多的数据，能利用海量的服务器资源捣腾海量的数据。

我的老朋友林世飞先生和陈欢先生正是这样的幸运者，他们刚毕业就加入了腾讯，在腾讯多个部门工作学习锻炼过，每天接触的都是上亿用户的数据处理。而现在他们又在腾讯发展最快速的部门——社交与效果广告部，负责大数据处理和分析的相关工作，利用成规模的 Spark 集群来做海量数据的处理，根据用户的画像给用户推荐个性化的广告。除此之外，林世飞和陈欢本身有着爱钻研的性格，他们成立了学习小组，拿着各种大数据技术的"锤子"，敲打着各种内外部的大数据"钉子"，来仔细分析、对比各种技术的差异点以及特性。在此基础上，他们总结成书，从运行原理到实际案例分析，覆盖了流式计算、数据仓库、图计算、机器学习等算法应用。在我看来，这应该是国内第一本兼顾基础，针对 Spark 的典型应用都做了案例讲解的实践书，对于创业公司快速搭建大数据系统，以及科研院校学生的毕业设计的实战项目帮助意义很大。

丰富的大数据实践经验、对大数据深入的思考，这是本书的两大特点，也是林世飞先生和陈欢先生严谨工作、对技术深入研究的体现，希望他们付出的心血能为每一位读者带来价值，使大家深入地了解 Spark 的工作原理、掌握好 Spark 的优势，为日常工作创造更大的价值。

季昕华
UCloud 联合创始人兼 CEO

序　　二

从 2003 年开始，谷歌先后发表了关于 GFS、MapReduce 和 BigTable 的 3 份重量级论文。在 Apache 软件基金会和雅虎等互联网公司的支持下，人们在参考谷歌的论文基础上实现了大量的开源服务框架（包括著名的 Hadoop、Hive 等重量级产品），特别是开源项目 Hadoop 的出现揭开了在 IT 行业，特别是互联网行业内大规模使用大数据技术的序幕。随着最近几年海量数据的持续增长和计算机硬件的发展，越来越多的新架构也涌现出来。从 2010 年开始，美国加州大学伯克利分校陆续提出了多份 RDD（Resilient Distributed Dataset，弹性分布式数据集）相关的论文，并随之推出开源的 Spark 框架。对比传统的 Hadoop，拥有深厚学术界背景的 Spark 把以往的 MapReduce、流式计算、机器学习算法等模型全部统一起来，让数据挖掘和机器学习的门槛大大降低，从而加速了大数据技术在各个行业和产品里面的普及。

手机 QQ 浏览器的大数据开发模式同样符合这种演进趋势。2011 年，我们主要基于 MapReduce 模式和 Hive 进行一些海量数据的分析和统计工作，而数据挖掘算法依然沿用传统的 C++和 MPI 框架方式。随着这两年 Spark 版本的不断更新和功能的逐步强大，我们已经开始用 Scala 和 Spark 自带的 MLlib 实现多个数据挖掘模型。由于使用了统一的 RDD 和 MLlib，我们可以快速实现各种算法模型，并在它们之间更灵活地进行切换和对比，这也体现了互联网快速迭代开发的效率和 ABTest 的思维模式。

最近的一年里面，世飞负责的广点通项目和手机 QQ 浏览器有深入的技术和产品层面的合作，广点通的广告推荐投放在手机浏览器里面也取得了非常好的效果。在工作交流和讨论的过程中，我深深体会到世飞在数据挖掘算法上的深厚功力，以及在 Spark 技术上追求极致的精神。这次有机会拜读世飞和陈欢所著的书稿，最大的体会是与市面上大量的手册型图书不同，本书不仅可以让读者逐步掌握 Spark 的核心概念和流程，而且得以吸取大数据业务特定场景下的实战经验。这些真正经过海量用户考验的行业案例，对于大数据的学习人员来说都传递着非常宝贵的经验。最后，衷心希望本书可以在为每一位读者带来 Spark 底层技术知识的同时，更好地让 Spark 普及到更多的大数据应用场景当中。

徐羽
手机 QQ 浏览器总监、腾讯 T4 技术专家

前　　言

近些年，大数据的概念非常火，且许多行业已经开始从大数据中获益，比如广告业。越来越多的企业也正在尝试使用大数据的平台和方法来解决一些现实问题，不过，在这个过程中难免会遇到困难。因此，本书旨在提供一个非常具有可操作性的指引，帮助读者亲身体验大数据的魅力。

Spark 是最近几年开始流行的一种通用分布式大数据计算平台，无论是性能、功能还是易用性，都在众多大数据技术中名列前茅，可以很好地解决大数据领域的许多常见问题。本书循序渐进地介绍 Spark 的基本概念、核心思想、部署、开发，并提供多个典型场景的解决方案，试图让即便是零基础的 Spark 读者也可以从中受益，并让对 Spark 已有所知的读者可以更深入地了解其运行机制及精髓。

没有大数据平台的支持，普通海量数据的计算分析极端低效

笔者 2008 年毕业后即进入腾讯公司工作至今，刚进公司时感觉真是“一夜暴富”，因为名下的服务器有上百台，日常工作除了后台应用程序的开发，还要担负一些运营工作。然而，维护上百台机器上产生的日志文件就是一个非常令人头痛的问题。虽然每台机器都有 TB 级的硬盘空间，但还是经常爆满，我们经常半夜被电话吵醒起来删文件。此外，产品人员经常提出一些数据统计分析的需求，然而数据分布在上百台机器上，统计效率非常低下。

面对这样大量数据的统计，最常用的一种处理模式是，选择一个字段对数据进行散列化，然后按散列值的不同分别存储在单独的 MySQL 数据库表中，而这些 MySQL 节点分布在上百台机器上。需要统计的时候，我们再写个脚本分别去各个 MySQL 节点上取数据，或者简单处理一下，然后再汇总到一台机器上做处理。

这种方法简单易行，但问题很多。最大的问题是计算过程复杂，开发统计脚本的时间以天计，因为原本一个 SQL 就可以搞定的事情，现在需要分多步进行，且中间还需要将大量数据在多个节点上进行传输。比如这个简单的案例，它需要对数据按 IP 进行分布统计，计算过程至少需要 3 步：

(1) 去各个 MySQL 节点上执行 SQL，按 IP 汇总统计；
(2) 数据汇总到一个节点上，导入 MySQL；
(3) 在汇总的 MySQL 节点上进行统计。

而如果数据没有拆分存储，一个 SQL 就可以搞定。这只是一个最简单的例子，如果统计更复杂一些，这样的“计算–汇总–计算”的过程可能需要进行多次。

这种方法带来的第二个问题是散列分布不均匀导致的计算瓶颈和资源浪费。为了方便查数据，散列的规则设计不能太复杂，一般是取某个整数的十进制数值的前 3 位或末 3 位。这样的好处是日常定位数据位置很方便，缺点是数据分布不均匀，总有一些节点会承载非常多的数据，而另外一些节点又承载非常少的数据。承载数据多的机器会成为整体计算的瓶颈，而承载数据少的机器总是空闲，从而造成资源的浪费。

所以整体看来，这种方法的效率非常低下，计算本身耗时虽然不多，但开发过程相当费时，如果逻辑更复杂，甚至需要一周时间。

Hadoop 和 Hive 的组合，可以解决大数据计算的许多常见问题

后来我们引入“Hadoop + Hive”的解决方案，问题改善了很多。数据全部存储在 Hadoop 上，在各节点上均匀分布。在 Hive 上创建表并导入数据，设置好分区，这样虽然数据的存储还是分布式的，但从数据统计角度来看像是一个数据库，而不是之前的上千个数据库表。这样，开发者只需要关注自己的业务逻辑，而不需要关注底层数据的存储、分区等信息，开发效率大大提升。虽然 Hive 的计算效率不是很高，运行时间可能是“脚本 + MySQL”方式的数倍，但一般也只需要几十分钟，而节省的开发成本是以“天”计的，整体效率大大提升。

然而，随着数据量的继续快速上升，以及大家对数据的关注度的提高，Hive 平台上的计算量越来越大，人们对于计算的需求也在变化。具体来讲，一是统计规则越来越复杂，原来几行 SQL 代码可以搞定的统计越来越少，新的统计需求动辄需要上百行 SQL 代码才能完成，而且量越来越多；二是越来越多的计算是 SQL 完成不了的，需要借助机器学习算法或图计算来进行。

这些新需求带来的挑战逐渐让“Hadoop + Hive”方式有些吃不消，一些问题逐渐暴露出来：一是性能不够高，一是简单的 MR 计算模式支持的应用场景非常有限。

Spark 出现之后，性能、功能、易用性都有质的提升

Hadoop 的这些问题暴露之后，业界也有一些针对性的优化方案。但 Spark 走得更远，在参考 MR 的基础上重新实现了一套通用计算框架，性能提升 10~100 倍，尤其是在机器学习等复杂迭代计算的场景下，性能提升最明显。另外，使用场景除了 SQL 之外，Spark 还支持类似 Storm 的实时流式计算和图计算，机器学习库也更丰富，为开发带来了极度的便利。因此，越来越多的技术人员开始使用 Spark 来替代 Hadoop 的 MR 计算，用 Spark SQL 来替代 Hive，但存储依然使用 Hadoop HDFS。

本书会详细介绍 Spark，值得注意的是，Spark 在设计伊始就充分考虑到了用户体验问题。

Spark 使用 Scala 语言开发，支持函数式编程，让代码非常简洁。比如，Hadoop MR 编程中的示例程序 WordCount，在 Hadoop 下用 Java 实现时代码有上百行，而且还需要经过编写代码文件、编译、上传执行等过程，而 Spark 集成的 Scala 语言支持交互式编程，进入交互式模式后直接输入代码即可开始执行，并且代码只有区区几行：

```
text_file = spark.textFile("hdfs://...")
text_file.flatMap(lambda line: line.split())
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a+b)
```

交互式编程配合函数式编程，使得 Spark 编程非常容易上手。笔者曾经考虑要不要让 Spark 支持 bash 这类脚本语言，但最终觉得 Scala 足矣。本书附录提供了 Scala 语法的简单参考，相信有一定编程基础的读者会非常容易上手。

经常有人会问：为何 Spark 会比 Hadoop 快那么多呢？我也曾反复思考过，在此抛砖引玉了。Spark 和 Hadoop 都是非常优秀的开源项目，代码实现及架构已经非常优秀了，差别应该不在实现层面。大多数人认为 Spark 比 Hadoop 快是因为大量使用内存，正因此，Spark 一开始被称作内存计算，但我认为还有一个更重要的原因，那就是核心数据结构。Spark 使用 RDD（弹性分布式数据集，读者可以将其想象成一个分布式的大数组）作为核心数据结构（相比之下 Hadoop 没有核心数据结构，只有 Map/Reduce 计算模式），在此基础上提供了许多计算函数，类似 Hadoop 下的 Map/Reduce 函数，但数量更多，还可以无限扩充。这样的好处是 Spark 的任务粒度更小，原先在 Hadoop 下一个 Map 或 Reduce 实现的功能，在 Spark 下可能会被拆分成多个 Job。如果把 Hadoop 中的任务比作罐子里面的石头，那么 Spark 的 Job 就是罐子里面的碎石子，因此相同罐子可以装得更多。Spark 这样细粒度的任务调度，再配合上内存的充分利用，最终让性能提升了一个台阶。

还有一个有意思的现象：大数据领域的技术很多是基于 Java 开发的，Hadoop、Hive、HBase 都是，Spark 使用的 Scala 最后也是转化成 JVM 字节码运行。Hadoop 虽然被诟病性能低，但也没有基于 C++的开源版本出现，究其原因，除了大家经常提到的这些开源系统的创始人喜欢 Java，以及 Java 对于跨平台的强有力支持，我猜测可能还有如下几个原因。一是因为 Java 在大型系统开发方面的便利性，有非常多的库可以使用。这样一来，在系统最初的探索阶段可以节省大量人力，而 Spark 使用的 Scala 语言因为支持函数式编程，代码量进一步精简了好几倍，比如 Spark 中的 SQL 优化执行引擎只用 2000 行左右的代码就实现了，而且表现接近于手写代码，完全受益于语言在开发上的便利性。二是因为大数据计算场景下，系统的瓶颈经常不在于计算，而是网络传输和磁盘读写，此时 CPU 反而不是瓶颈，所以无论 Hadoop 和 Spark 都在这方面做了大量优化，语言性能方面的弱势不那么明显了。比如 Hadoop 和 Spark 中最重要的 shuffle 机制，就是为解决网络传输而设计的，而且不断优化，每次优化都可以在整体上提升性能。三是在 Hadoop 和 Spark 这类通用计算平台上都遵循一个理念，就是“宁可移动计算，不要移动数据”，这正是 Java 语言强大的序列化功能发威的时候。当然 C++也可以实现，性能可能会更优，但开发成本可能不是一般小公司能承受得了的，也只有谷歌这样技术、财力雄厚的公司才有可能实现。事实上，Hadoop

的理论基础也来自谷歌的论文。

大数据平台众多，我们要怎么选呢

大数据热门之后出现了非常多的大数据技术，有 Hadoop、Hive、HBase、Flume、Kafka、Lucene、Spark、Storm 等，还有很多 NoSQL 技术可以归入大数据，比如 MongoDB、CouchDB、Cassandra 等。有人列举过大数据技术族谱，其中至少上百种技术。面对如此多的技术，我们往往会产生疑惑：到底该如何选择？这里有一个建议：在充分理解自身需求的基础上，挑选最合适的。

比如要实现这样一个任务：有 10 亿条用户画像数据，里面记录了用户的性别、年龄、地域等信息，输入一个用户包，只包含用户账号，数量平均百万级，最多 1000 万，希望查询这些用户的性别、年龄、地域等属性的分布信息。

面对这样的问题，如果我们贸然找一个开源系统部署上，刚开始的学习成本高不说，即便可以用，最后也不一定能证明这个方案到底是好还是不好。因为从来没有最好的方案，只有当前最合适的方案，不但要跟其他方案比，还要跟问题本身的需求比。比如，一开始就有一些有经验的前辈建议使用 Solr 或 ElasticSearch，这两个系统固然非常强大，但大家都知道是为搜索引擎这样的场景设计的，而不是为这个问题场景设计的，中间难免有些匹配度的问题，不一定最合适。而且，如果不分析一下面对的问题场景，那么在研究 Solr 或 ElasticSearch 时也没有针对性，只能“全面开花”，结果必然费时费力，可能还没有好的结果。要知道，单 Solr 的文档就有几百页，全部看完至少需要一周时间。

因此我们先来分析一下当前的问题。如果所有数据都在数据库中，那么这个计算可以用 SQL 实现（以统计“性别”属性为例，其他与此类似）：

```
select
    性别, count(1)
from
    用户号码包表(1000W) join 画像表(10 亿)
group by
    性别
```

显然计算的瓶颈在于 join 操作，join 之后需要处理的数据量从 10 亿下降到 1000 万，这可以大大降低后续的聚合统计的成本。而对于 join 操作来说，如果两边的数据都已经排好序，那么 join 的算法复杂度只是 $O(n)$，非常低。由于 10 亿条画像数据更新频率很低，我们可以提前做好排序预处理，因此最理想的计算过程可以分为这么几步：

(1) 对输入的用户号码包进行排序；

(2) 排序后的号码包与已经有序的画像数据进行 join，筛选出号码包的属性；

(3) 对筛选后的少量数据进行聚合统计。

有了对问题的理解，我们再简单对比一下各种方案的优劣（使用相同的机器，集群数量为10台），参见下表。

方　　案	实现过程	总用时	缺　　点
Hive集群 （10台）	(1) 画像数据和号码包都存储为一张表； (2) 对两张表进行join操作，筛选出号码包的属性； (3) 对每个属性进行统计	> 1小时	机器资源占用多，维护成本高
Spark SQL （10台）	同Hive	10分钟	
Spark MR （10台）	思路同上，但可以让画像数据提前排好序，并且两张表使用相同的分区策略，这样可以提升join的效率	8分钟	
Bash (单机单核)	(1) 对号码包进行排序； (2) 号码包文件与有序的画像文件进行join，筛选出号码包的属性； (3) 对筛选出的结果按维度分别统计	12分钟	单点
Bash （单机多核）	思路同上，但把大文件拆成小文件，并发执行	3分钟 磁盘替代CPU成为瓶颈，如果有多个物理磁盘，性能可以进一步提升至1分钟	
从零开发一个专有分布式系统 （3~10台）	思路同上，但所有的操作都在内存中，可能需要多台机器	< 0.5分钟	开发成本太高 对开发人员要求高
Solr （3台机器）	(1) 设置集群为多shard结构，画像数据提前导入； (2) 新的画像数据作为新的字段导入； (3) 使用join操作和Facet特性对各个维度分别统计	1分钟	没有筛选，每个维度的查询都搜索了所有10亿画像数据 Facet特性必须指定一个UniqueKey，导致号码包导入时需要多处理一个UniqueKey字段

通过分析不难发现，有很多技术可以解决这个问题，连最不起眼的 bash 都可以，而且单机的性能就接近甚至超过 10 台 Spark 集群的性能。如果单独开发一个专有系统，性能更是会得到指数级提升。

然而，通过这些对比依然不足以确定我们的最终方案，因为对于需求本身还有很多疑问，比如系统的访问频率、扩容要求、需求变更，甚至还有团队的研发能力等。如果回答了这些问题，那么最终的决策只是水到渠成的事情。

本书内容

全书共分 8 章，外加一篇附录。前 4 章介绍 Spark 本身，包括部署、工作机制、内核等。全书的重点在第 5 章~第 8 章，每章不但深入浅出地介绍 Spark 的一个功能模块，而且包含一个实战项目，项目利用国内互联网的真实数据为案例，搭建一个产品和平台输出。这些例子每个都可以是一个独立的大项目。这在 BAT 等公司中，动辄都是有几十上百人的项目团队，而笔者基于 Spark 快速搭建了一个原型，为创业者提供了很好的入门示例。此外，书中还详细介绍了各种可能遇到的实战问题，比如大数据环境下的配置设置、程序的调优等。随书带一键安装脚本，以便为初学者提供很多帮助。

- 第 1 章概述大数据的发展状况，以及 Spark 的起源、特点、优势、未来等。
- 第 2 章介绍 Spark 部署和编程。作者首先带领读者在本地单机下体验 Spark 的基本操作，然后部署一个包括 ZooKeeper、Hadoop、Spark 的可实际应用的高可用集群。并且，这一章还介绍了笔者合作开发的一个自动化部署工具，最后介绍了 Spark 编程的基础知识，以及如何打包提交至集群上运行。
- 第 3 章介绍 Spark 底层的工作机制，包括调度管理、内存管理、容错机制、监控管理以及 Spark 程序配置管理，这对理解 Spark 程序的运行非常有帮助。
- 第 4 章深入 Spark 内核，并结合源码，介绍了核心结构 RDD、RDD 对象的 Transformation 和 Action 操作是如何实现的、`SparkContext` 对象及初始化过程、DAG 调度的工作流程。了解这些内容可以帮助读者编写出高质量的 Spark 程序代码。
- 第 5 章介绍 Spark SQL，可以代替 Hive，用于搭建一个企业级的数据仓库。案例基于淘宝的电商数据建立电商数据仓库，并以日常运营工作为例，通过电商数据库分析电商运营中的各类问题。
- 第 6 章介绍 Spark 实时流式计算，类似于 Storm，但吞吐量方面更有优势。案例是基于一个站点的 Web 日志建立一个类似百度统计的实时统计系统，是各种实时系统典型的参考例子。
- 第 7 章介绍 Spark 的图计算。案例基于新浪微博 2000 万的关系链数据，讲解了如果利用图计算来实现社交关系链的挖掘，比如闺蜜的发现、粉丝团伙的发现等。
- 第 8 章介绍 Spark 的机器学习库。案例基于某个搜索引擎的点击日志，建立了一个搜索广告点击率预估系统。广告点击率预估是各家互联网系统的核心系统，公开的实战项目不多。
- 附录为 Scala 语言参考，方便第一次接触 Scala 语言的读者快速上手，体验 Spark 编程。

致谢

本书由陈欢和林世飞一起合作编著，大家年龄相仿，志同道合，牺牲了很多陪家人的时间才完成本书的撰写，非常感谢家人的支持。还要特别感谢腾讯企业云的“攻城之地”活动提供免费服务器，感谢李冠灿为服务器使用提供很多便利，感谢李学凯、余衍炳等人审阅了部分章节，提

供了许多宝贵意见。感谢王流斌在如何使用 KDD2012 数据集方面提供了不少建议。感谢腾讯公司提供的学习和成长环境，以及各位领导对于我们撰写本书的鼓励和支持。感谢 UCloud 的 CEO 季昕华提供宝贵建议，并撰写“推荐序”。成稿期间，图灵出版公司的王军花编辑和毛倩倩编辑给予了极其详细的改进意见，在此表示由衷的感谢。

由于作者水平有限，书中难免有错误和不当之处，欢迎专家和读者给予批评指正，来函请发至 sparkbestpractices@qq.com。

2016 年 2 月

目　　录

第1章 Spark与大数据

数据只是工具，最终还是要用于创造价值，大数据只是一种新的实践。

1.1 大数据的发展及现状

大数据是一个很热门的话题，但它是从什么时候开始兴起的呢？

“大数据”（big data）这个词最早在UNIX用户协会的会议上被使用，来自SGI公司的科学家在其文章“大数据与下一代基础架构”（“Big Data and the Next Wave of InfraStress”，参见http://static.usenix.org/event/usenix99/invited_talks/mashey.pdf）中用它描述数据的快速增长。那么，怎样才叫“大数据”呢？META集团（已被Gartner收购）的一份报告提出大数据的特征“3V”，即大量（volume）、多样（variety）、快速（velocity）。如今，大家更普遍地使用“4V”来描述大数据的特征，即增加了第四个V——价值（value）。

从数据量上来看，人们通常认为当一个计算单元容纳不下要处理的数据，就是在面对大数据了。其实早在电脑被发明之前，我们做的人口统计、气象预报等工作就属于大数据范畴了。下面，让我们看看大数据时代所面临的问题。

1.1.1 大数据时代所面临的问题

我们拿搜索引擎举例。搜索引擎需要把网上的多数网页抓取下来进行分析，并建立索引，这样在我们搜索一个词的时候，它才能在毫秒级别返回结果。我们假设每个网页平均的大小为20 KB，大概有价值的中文网页有200亿个页面，不过考虑到压缩情况，大概总共有400 TB的大小。那么，一台计算机以30 MB/s~35 MB/s的速度从硬盘读写，大概需要4个多月，这还不包括在这些数据上做复杂的分析工作。因此，我们通常需要一个计算机集群来完成相关的工作。

有人可能会说，我们的计算机处理能力在以指数级增长，这样机器能够处理更多的数据。是的，这也是事实，但是这些计算机生产数据的能力也在以指数级增长。而且，每人拥有的设备数在增加，如今除了个人电脑，越来越多的人拥有智能手机、平板电脑、智能家居、穿戴设备等。有数据表明，90%的数据是过去两年左右时间内产生的。

我们面对的最直接的问题是如何存储和计算大数据。接下来仍以搜索引擎作例子。假设每台机器存储400 GB的数据，那么我们需要1000台机器来存储这些数据。下面来分析下1000台服务器的故障率。

X表示一台服务器是否正常工作，$X=0$表示正常工作，$X=1$表示不正常工作。服务器一个月内发生故障的概率是ε，$P(X=0)=1-\varepsilon$，那么1000台电脑都正常工作的概率是$(1-\varepsilon)^{1000}$。假设故障率$\varepsilon=0.001$，所有1000台服务器1个月内正常工作的概率只有0.37，不到一半。

所以，随着数据的增加、机器的增加，处理数据的技术也变得越来越复杂了。为了解决这些问题，各种大数据解决方案诞生了。如今，程序员在100台机器上编程和在1000台上面编程没有太多区别，不需要考虑容错性、并发等问题。

随着云计算技术的发展，不仅大数据处理的复杂度降低了，成本也大大降低，云计算服务商的IaaS（Infrastructure as a Service，基础设施即服务）支持按天付费，而且可以动态按需扩容。在云计算服务商的帮助下，如今一个小创业公司都能够快速开发和部署大数据应用。云计算服务商的国外领导者是Amazon AWS，国内主要是阿里云、腾讯云和UCloud。

另外一个困扰我们的问题是：数据这么多，我们要存储哪些数据，以及存储多长时间的数据。通常我们认为，从一个字节可以获取多少价值，存储它又要花费多少费用，如果两者的比值大于1，就值得存储更多数据。大数据还有一个特点，就是存储单条数据的价值可能不大，拥有更多数据时，就有价值了。举一个简单的例子。如果我们只有一个用户的访问日志，显然没有太大价值，但是如果拥有全站的用户访问日志，我们就有可能对数据进行分析，从而发现潜在规律和趋势等。

解决了大数据存储和计算的问题之后，如何对数据进行分析呢？这是一个复杂的技术。Spark的很多库就是为了解决不同场景下的分析任务而存在的，比如MLlib库就是为解决各种机器学习问题开发的库，这些问题包括分类问题、回归问题、聚类等。GraphX就是为了分析社交网络等应用开发的库。

大数据时代面临的另外一个问题就是大数据交易，数据本身已成为一种可以交易的商品。2015年9月5日国务院印发了《促进大数据发展行动纲要》，纲要提出要建设公共数据资源开放的统一开放平台，“2020年底前，逐步实现信用、交通、医疗、卫生、就业、社保、地理、文化、教育、科技、资源、农业、环境、安监、金融、质量、统计、气象、海洋、企业登记监管等民生保障服务相关领域的政府数据集向社会开放”，以及“到2020年，培育10家国际领先的大数据核心龙头企业，500家大数据应用、服务和产品制造企业”。

1.1.2　谷歌的大数据解决方案

谷歌的搜索引擎是搜索引擎界的领导者，很重要的原因之一就是谷歌在大数据技术方面领先。2003年开始，谷歌先后发表了GFS、MapReduce、BigTable等几篇论文，其中2004年，谷歌的Jeff等发表了论文“MapReduce: Simplified Data Processing on Large Clusters”，提出了大数据分析的范式MapReduce。虽然MapReduce范式在函数式编程中为人所熟知，但是该论文提供了在集

群环境中该范式的可扩展性实现，使得程序员可以利用机器集群处理更多数据，而不需要考虑容灾和并发等问题。谷歌的三大论文引发了人们在大数据领域的大量研究，直接导致了Hadoop的出现——MapReduce范式的开源实现。如今Hadoop，包括MapReduce与分布式文件系统（HDFS），已经成为数据处理的事实标准。大量的工业界应用，例如腾讯、百度、阿里巴巴、华为、迪斯尼、沃尔玛、AT&T都已经有自己的Hadoop集群。

MapReduce能做的事情很多，包括实现各种机器学习算法（参见http://machinelearning.wustl.edu/mlpapers/paper_files/NIPS2006_725.pdf）。

但MapReduce不是唯一的大数据分析范式，有一些场景是不适合使用MapReduce的，比如处理网状的数据结构时，这要求能够处理顶点和边的增加和减少操作，并在所有节点进行运算等。

典型的场景就是在搜索引擎中链接地图计算和社交网络分析。谷歌也建设了基于图的计算系统Pregel，允许连通的节点之间互相交换信息。Pregel基本运算是节点之间的超级步骤（superstep），每个顶点都有一个用户自定义的计算函数和值，所有的边都可以并行计算，顶点可以通过边来发送消息并与其他顶点交互数据，可以聚合所有节点的信息，计算最大最小值等。Spark上也能够很方便地支持图计算应用。事实上，最初AMPLabs团队仅用数百行代码就开发出了整个Pregel，如今图计算库GraphX已经是Spark主要的库之一。

当然，并不是说谷歌出现前就不存在大数据。我们使用的很多方法和技术都是过去已有的，比如分布式系统广泛应用的Paxos算法，是莱斯利·兰伯特（Leslie Lamport）于1990年提出的一种基于消息传递的一致性算法。

1.1.3　Hadoop 生态系统

Hadoop是谷歌大数据解决方案的开源实现，使用Java语言开发，其核心主要是两部分：分布式文件系统（HDFS）和MapReduce。为了处理更多的应用场景，在此基础上，经过业界巨头雅虎等，以及开源界其他力量的努力，人们建设了很多其他的重要系统，这里我们简单介绍下。

Hive是在HDFS和MapReduce上提供一个类似于SQL风格的抽象层，非常容易上手。

用户可以用数据库、表的概念来管理数据，使用SQL来访问、计算，不需要写MapReduce程序。SQL语法非常类似于关系型数据库，支持常见的`Select`、`Join`、`Group by`、`Insert`等操作。

HBase是基于Hadoop的非关系型数据库，具备分布式、可扩展的特点，支持在几十亿行、数百万列的一张大表上进行实时、随机地读写访问。典型场景有各种数据仓库，比如淘宝用户历史订单查询等。

ZooKeeper是提供分布式应用程序协调服务的系统，是谷歌的Chubby一个开源的实现，是Hadoop和HBase的重要组件。比如，Spark为了保证高可用，同时运行多台Master节点，但只有一台是活跃的，其他都处于热备状态，通过ZooKeeper可以协调选择出当前活跃的节点，当这个活跃节点异常时，再从剩下的热备节点中重新选择一台活跃节点。

Hadoop是一个批处理系统，不擅长实时计算，如果需要实时或准实时的分析，可以使用Storm（Twitter）、S4（雅虎）、Akka等系统。另外，Hadoop也不擅长复杂数据结构计算，比如前面提到的图计算，可以利用的开源系统有GraphLab和Spark的GraghX库。

从Hadoop 2.0开始，Hadoop YARN将资源调度从MapReduce范式中分离出来。YARN已经成为通用的资源管理系统，可为上层应用提供统一的资源管理和调度，Spark支持部署在YARN管理的集群上。

在工业界大规模应用Hadoop生态的系统，还要面临部署、排错、升级等问题。为解决这些问题，降低Hadoop应用门槛，我们可以使用Hadoop商用解决方案的提供商的产品，目前比较成熟的提供商有Cloudera、Hortonworks和MapR。

Cloudera由Doug Cutting和Jeff Hammerbacher共同创建，为合作伙伴提供Hadoop的商用解决方案，主要包括支持、咨询服务、培训。Cloudera产品主要为CDH、Cloudera Manager、Cloudera Support。CDH是Cloudera的Hadoop发行版，完全开源，比Apache Hadoop在兼容性、安全性、稳定性上有增强。Cloudera Manager是集群的软件分发及管理监控平台，可以在几个小时内部署好一个Hadoop集群，并对集群的节点及服务进行实时监控。Hortonworks是雅虎与硅谷风投公司Benchmark Capital合资组建的公司，公司成立之初吸纳了大约30名专门研究Hadoop的雅虎工程师，这些工程师均在2005年开始协助雅虎开发Hadoop，贡献了Hadoop 80%的代码。Hortonworks的主打产品是Hortonworks Data Platform（HDP）。

目前CDH和HDP在国内普及率不高，国内巨头使用的都是在Apache Hadoop开源版本上自己修改的定制版本，每家公司的集群规模也都有几千台。国内的公司也开始给开源社区同步部分修改，根据不完全统计，华为有4个Hadoop committer，小米有3个HBase committer，腾讯有一个Storm的committer，阿里巴巴有一个Spark的committer等。相信往开源社区贡献代码是未来的一种趋势。

1.2　Spark 应时而生

Hadoop MapReduce可以解决很多通用计算的问题，但在某些场景下效率不高，Spark就是在这个场景下诞生的。

1.2.1　Spark 的起源

机器学习算法通常需要对同一个数据集合进行多次迭代计算，而MapReduce中每次迭代都会涉及HDFS的读写，以及缺乏一个常驻的MapReduce作业，因此每次迭代需要初始化新的MapReduce任务。这时，MapReduce就显得效率不高了。同时，基于MapReduce之上的Hive、Pig等技术也存在类似问题。

Spark作为一个研究项目，诞生于加州大学伯克利分校AMP实验室（Algorithms, Machines, and

People Lab）。AMP实验室的研究人员发现在机器学习迭代算法场景下，Hadoop MapReduce表现得效率低下。为了迭代算法和交互式查询两种典型的场景，Matei Zaharia和合作伙伴开发了Spark系统的最初版本。2009年Spark论文发布，Spark项目正式诞生，在某些任务表现上，Spark相对于Hadoop MapReduce有10 ~ 20倍的性能提升。2010年3月Spark开源，且在开源社区下发展迅速。2014年5月，Spark 1.0正式发布，如今已经是Apache基金会的顶级项目了。

1.2.2 Spark 的特点

Spark刚出现时，常常被大家概括为内存计算，这不是没有缘由的。在典型应用中，Spark读取HDFS中的文件，加载到内存，在内存中使用弹性分布式数据集（Resillient Distributed Dataset，RDD）来组织数据。RDD可以重用，支持重复的访问，在机器学习的各个迭代中它都会驻留内存，这样能显著地提升性能。即使是必须使用磁盘进行复杂计算的场景，Spark也常常比Hadoop MapReduce更高效。

Spark频频在效率上表现出色，其官方博客显示，Spark在最近的排序大赛（https://databricks.com/blog/ 2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html）中创造了新纪录，参见表1-1。

表1-1　Spark的新纪录

	Hadoop MR	Spark	Spark
数据规模	102.5 TB	100 TB	1000 TB
耗时	72 min	23 min	234 min
节点数	2100	206	190
核数	50 400（物理的）	6592（虚拟的）	6080（虚拟的）
磁盘吞吐量	3150 GB/s（估计）	618 GB/s	570 GB/s
环境	专用数据中心，10 Gbit/s	虚拟（EC2）10 Gbit/s网络	虚拟（EC2）10 Gbit/s网络
排序效率	**1.42 TB/min**	**4.27 TB/min**	**4.27 TB/min**
节点排序效率	**0.67 GB/min**	**20.7 GB/min**	**22.5 GB/min**

最新的结果可以关注（http://sortbenchmark.org/），最近提交的截止时间是2015-9-1。

Spark一直寻求保持Spark引擎小而紧凑。Spark 0.3版本只有3900行代码，其中1300行为Scala解释器，600行为示例代码，300行为测试代码。即使在今天，Spark核心代码也只有约50 000行，因此更容易为许多开发人员所理解和供我们改变和提高。

Spark是一个通用计算框架，包含了特定场景下的计算库：Streaming、SQL、MLlib、Graphx等。除了支持常见的MapReduce范式，它还能够支持图计算、流式计算等复杂计算场景，在很大程度上弥补了Hadoop的不足。

此外，Spark的接口丰富，提供了Python、Java、Scala等接口，文档清晰，为初学者提供了便利的上手条件。

在容错方面，Spark也有自己的特色。容错机制又称“血统”（Lineage）容错，即记录创建RDD的一系列变换序列，每个RDD都包含了它是如何由其他RDD变换过来的，而且包括如何重建某一块数据的信息。由于Spark只允许进行粗粒度的RDD转换，所以其容错机制相对高效。Spark也支持检查点（checkpoint）的容错机制。容错的机制和原理将在后面详细阐述。

Spark自带调度器，同时能够运行在Hadoop YARN集群、Apache Mesos上，可以很方便地和现有的集群进行融合。

Spark的输入支持本地存储、Hadoop的HDFS，以及其他支持Hadoop接口的系统：S3、Hive、HBase等。Spark还有一个优点，就是当RDD的大小超出集群的所有内存时，可以优雅地进行降级支持，存储在磁盘。

从成本的角度出发，由于Spark能适应多种应用场景，这样一个公司或者组织可能就不需要部署多套大数据处理系统，可大大减低学习、维护、部署、支持等成本。

1.2.3　Spark 的未来发展

Spark在过去的5年里发展迅速，社区活跃程度一点儿不亚于Hadoop社区。我们从Matei Zaharia在Spark五周年的总结中可以看到，Spark的contributor呈指数级增长，参见图1-1。

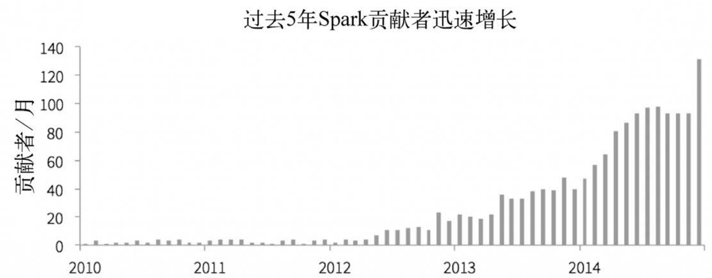


图1-1　Spark的贡献者在过去5年内的增长情况

Spark峰会（http://spark-summit.org）是Spark社区分享Spark各种应用的重要交流会。在Spark Summit 2015上，来自Databricks、UC Berkeley AMPLab、百度、阿里巴巴、雅虎、英特尔、亚马逊、Red Hat、微软等的数十个机构共分享了近100个精彩纷呈的报告。

目前，Databricks、英特尔、雅虎、加州大学伯克利分校是Spark主要的贡献者。

Spark 1.4中还发布了SparkR，引入了更友好的R语法支持，进一步扩展了Spark数据分析的应用场景。

工业界应用上，从Spark官方提供的用户列表可以看到，国内的BAT都在用Spark，完整的列表可以参考https://cwiki.apache.org/confluence/display/SPARK/Powered+By+Spark。Spark Summit 2015中，Databricks的CTO Matei Zaharia在Keynotes里指出Spark最大的集群来自腾讯，一共有8000个节点，单个Job最大分别是阿里巴巴和Databricks的1 PB。

如今，Cloudera、Hortonworks和MapR在内的大数据发行版也添加了Spark，Spark已经得到广泛认可，成为一种优秀的大数据处理技术平台。

传统道家思想把事物分为道、法、术、器4个层次。如果说Spark是大数据的一件利器，那么本书最多就算是指导使用Spark的“术”，然后解决大数据的问题，需要的更多是“道”“法”的探索。

第 2 章

Spark基础

2

本章带领大家体验一下Spark。首先，在虚拟机环境下体验所有基本操作，然后部署一个可供实际使用的高可用集群，之后会探讨如何编写Spark程序并提交至集群上运行。

2.1 Spark 本地单机模式体验

Spark提供了本地单机部署的体验模式，与真实集群相比，它们的最大差异只是计算资源较少，但用来学习或体验足够了。这里的演示都在Linux系统下，这也是考虑到Spark真实的部署环境绝大部分都在Linux下，而且Linux下强大的shell功能让我们操作起来更方便。

2.1.1 安装虚拟机

已经有Linux环境的读者，可以跳过本节，没有的话，可以参考本节在Windows桌面环境下安装一个Linux虚拟机。

虽然各大云平台都有免费的云服务器可以试用，但还是有诸多不便。比如，国内的腾讯云、阿里云、UCloud，试用期一般不到一个月。虽然国外的亚马逊提供一年的试用期，但需要填写信用卡支付信息，一不小心就可能因为流量或其他资源使用超标导致自动扣费，而且网络稳定性也是个问题。所以相比之下，本地虚拟机最方便，没有网络时也可以使用。

1. Linux发行版Ubuntu

安装虚拟机之前，首先选择一个Linux发行版。其实任何Linux发行版都可以，这里我们推荐使用比较流行的Ubuntu，其下载地址为http://www.ubuntu.com/download/。我们可以根据个人习惯选择纯shell操作的Ubuntu Server版本，或者带图形界面的桌面版。

2. 虚拟机工具VirtualBox

常见的虚拟机有VMWare Workstation、VMWare Player和VirtualBox，VMWare Workstation功能最强大但收费，我们可以选择免费的VMWare Player或者VirtualBox，这里以VirtualBox为例来演示安装。VMWare Player的操作也基本类似。

VirtualBox的下载地址为https://www.virtualbox.org/wiki/Downloads，截至2015年10月底时的最新版本是5.0.8，我们下载最新的VirtualBox 5.0.8 for Windows hosts，它已经支持到Windows 10了。其安装过程基本上是一直点击“下一步”，安装成功后的界面如图2-1所示。

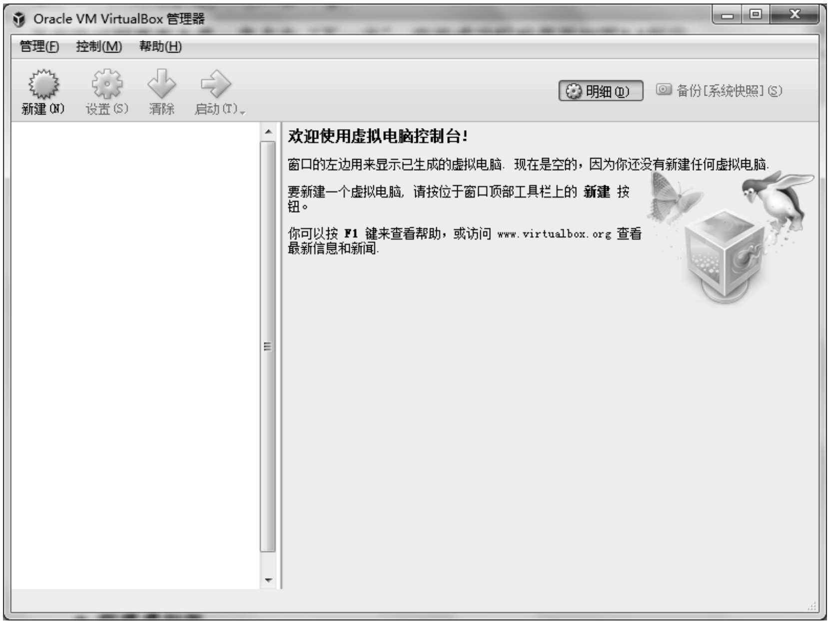


图2-1　成功安装VirtualBox后的界面

● **创建虚拟机**

点击主界面左上角的“新建”按钮，在打开的界面中选择Linux类型、Ubuntu (64-bit)版本，并取个名字，比如Spark Ubuntu，如图2-2所示。

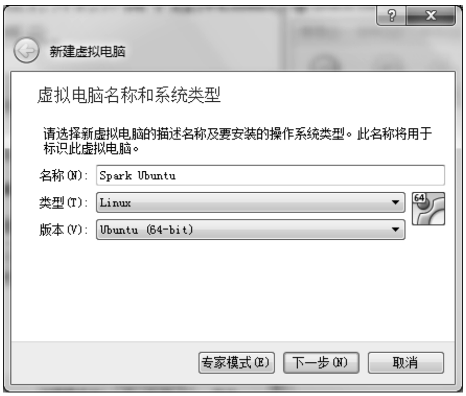


图2-2　新建虚拟电脑

然后一路点击“下一步”按钮，大部分使用默认配置就好，而磁盘空间这里可以稍微设置大一点，因为实际占用的本地磁盘是动态分配的，所以不会占用更多的磁盘空间，如图2-3所示。

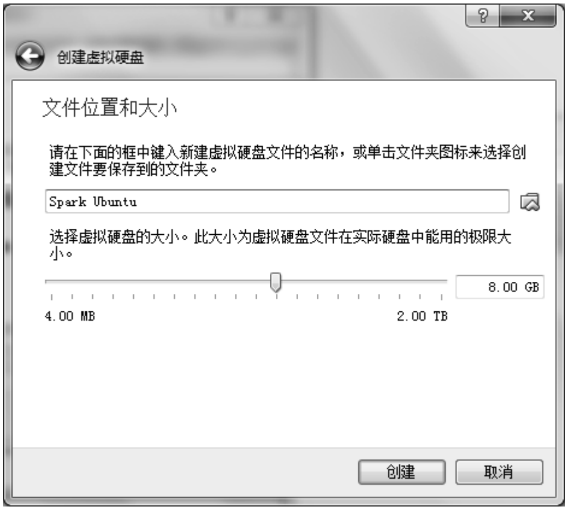


图2-3　文件位置和大小

创建好后，就可以看到Spark Ubuntu虚拟机了，如图2-4所示。



图2-4　创建好的Spark Ubuntu虚拟机

默认情况下，虚拟机可以访问外网，但宿主机（即当前的个人电脑）却无法访问虚拟机。为了后面操作方便，我们配置一下，让宿主机也能访问虚拟机。

选中Spark Ubuntu虚拟机，点击“设置”→“网络”，启用“网卡2”，连接方式选择“仅主机（Host-Only）适配器”，点击“确定”按钮保存即可，如图2-5所示。这样未来的虚拟机会有两块网卡，一块用于通过宿主机连接外网，另一块用于与宿主机通信。

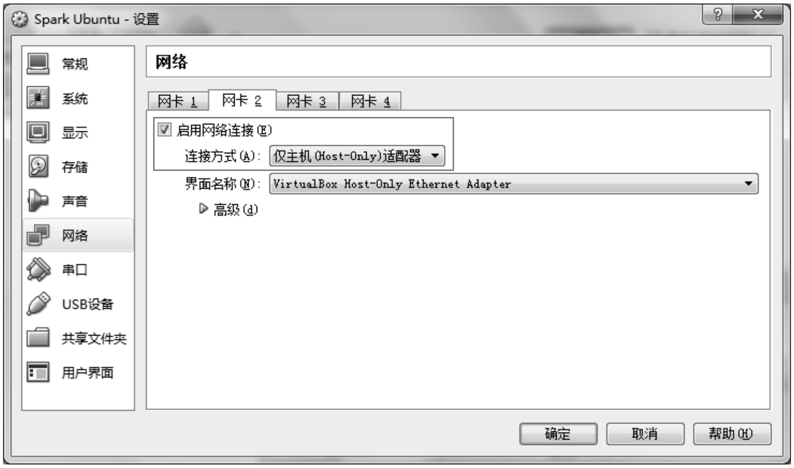


图2-5　设置宿主机可以访问虚拟机

需要说明的是，这一步也可以在安装完操作系统之后进行。

- **给虚拟机安装操作系统Ubuntu**

这时的虚拟机只是一台空机器，上面没有操作系统，我们需要为它安装操作系统Ubuntu Server LTS。

点击“启动”按钮，启动之前会提示我们选择一个启动盘，这里选择使用前面下载的镜像文件ubuntu-14.04.3-server-amd64.iso，如图2-6所示。

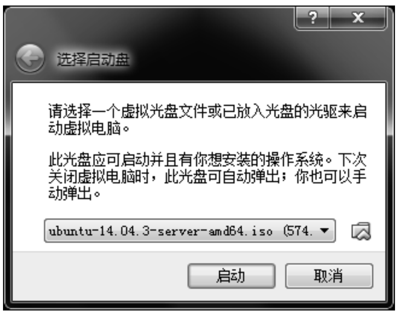


图2-6　选择启动盘

然后启动虚拟机，可以看到虚拟机已经运行起来了，如图2-7所示，而且会看到Ubuntu的安装界面，如图2-8所示。



图2-7　启动虚拟机

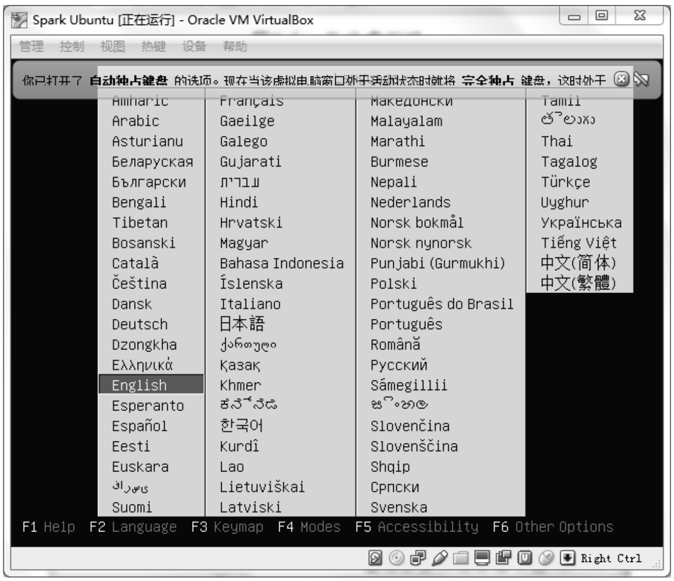


图2-8　Ubuntu的安装界面

这是虚拟机的界面窗口，点击鼠标后，个人电脑的鼠标、键盘将完全被虚拟机占有，如果不熟悉Linux，会感觉鼠标、键盘完全失灵，难免不安，不过没关系，退出的方式在窗口右下角有提示：按“右Ctrl”键即可退出。

在安装过程中，绝大部分配置使用默认值就好。当然，我们也可以选择自己偏好的语言。格式化磁盘时，需要手动选择Yes。唯一需要我们输入的地方是创建一个root之外的账号，并设置密码，如图2-9和图2-10所示。

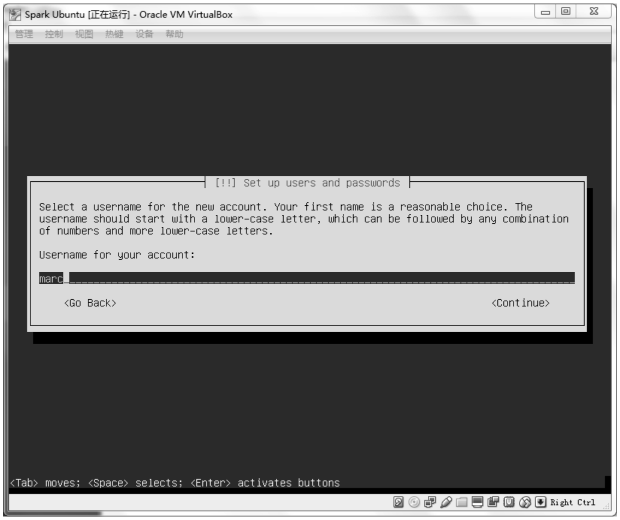


图2-9　添加账号（另见彩插图2-9）

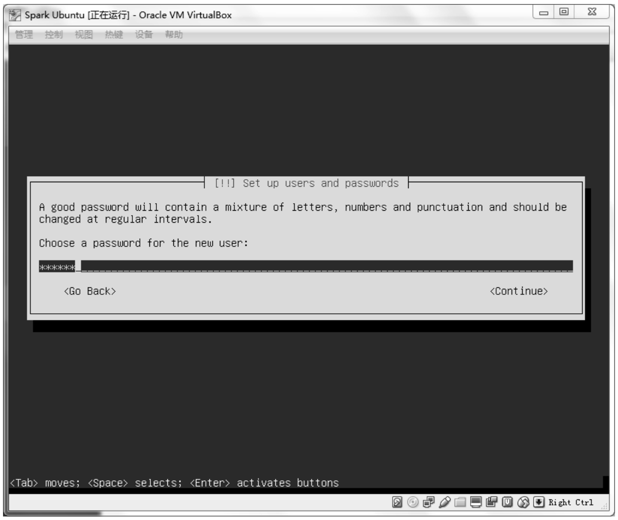


图2-10　设置密码（另见彩插图2-10）

在选择安装哪些服务时，我们选上OpenSSH server，此时会自动启动SSH服务，方便从本机访问虚拟机，如图2-11所示。当然，也可以事后安装服务。

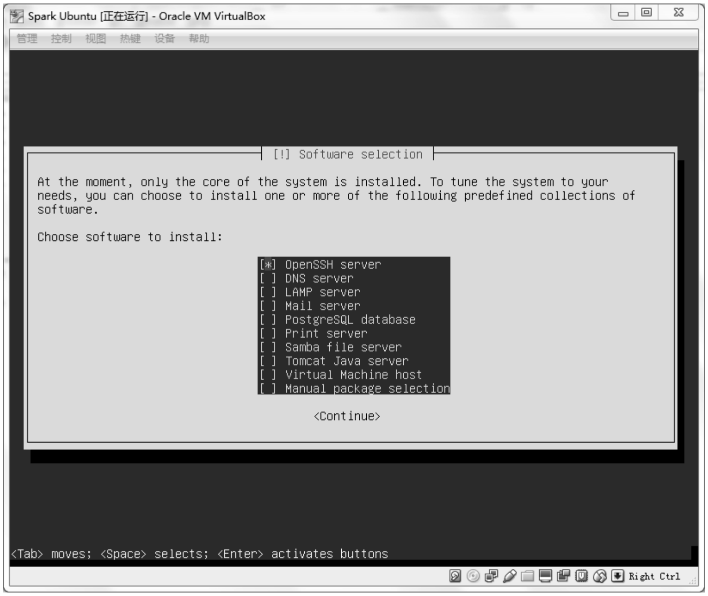


图2-11　选择OpenSSH server（另见彩插图2-11）

安装过程大概持续半小时，这视机器配置而略有不同。

安装完成后会自动重启，输入刚才创建的账号和密码之后，就可以登录进去了，如图2-12所示。

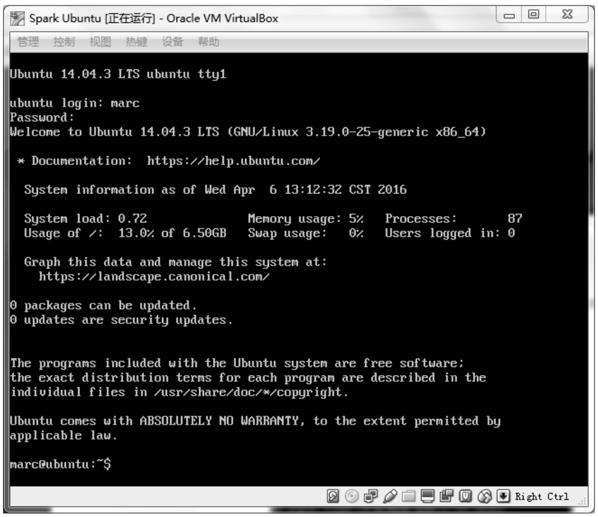


图2-12　登录成功界面

在Ubuntu系统下，我们可以通过`apt-get`命令安装各种程序，比如安装上传和下载文件的工具：

```
sudo apt-get install rpm alien lrzsz
```

3. 网络配置

通过`ifconfig`命令可以看到自己的地址，eth1的IP即我们可以从宿主机访问的IP，如图2-13所示。

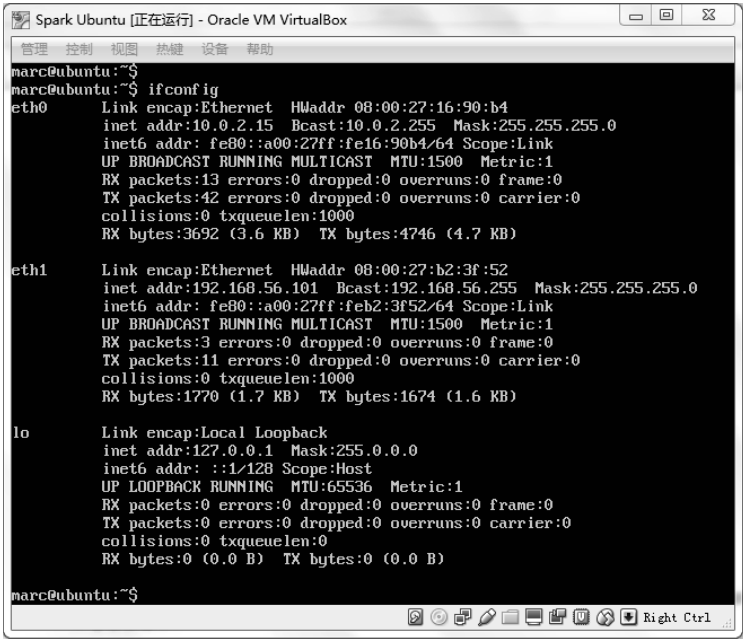


图2-13　eth1的IP

如果是安装完操作系统后才添加的网卡2，默认不会启动，可以执行下面的命令来启动网卡，并自动获取IP：

```
sudo ifconfig eth1 up
sudo dhclient
```

如果在安装Ubuntu时没有选择OpenSSH Server，可以手动安装：

```
sudo apt-get install openssh-server
```

这里的IP是通过DHCP服务自动分配的。为了避免虚拟机下次重启之后变更IP，我们可以配置虚拟机使用固定IP。编辑网络配置文件：

```
sudo vi /etc/network/interfaces
```

添加如下内容：

```
auto eth1
iface eth1 inet static
address 192.168.56.101
netmask 255.255.255.0
gateway 192.168.56.100
```

其中网关地址192.168.56.100来自VirtualBox全局设定中的网络配置，如图2-14所示。



图2-14　VirtualBox全局设定中的网络配置

重启网络服务后生效：

```
sudo /etc/init.d/networking restart
```

4. SSH客户端

直接在虚拟机平台的shell下操作不太方便，而且也不方便与宿主机的文件传输，所以要借助远程SSH连接工具。最好用的是SecureCRT，其功能强大，不过是商业版，但提供了30天免费试用版，足够我们体验Spark了。

当然，也有开源免费的可供选择，此时可以用Bitvise SSH Client，其官方网址为https://www.bitvise.com/，下载后安装并启动，按照图2-15输入IP和端口。

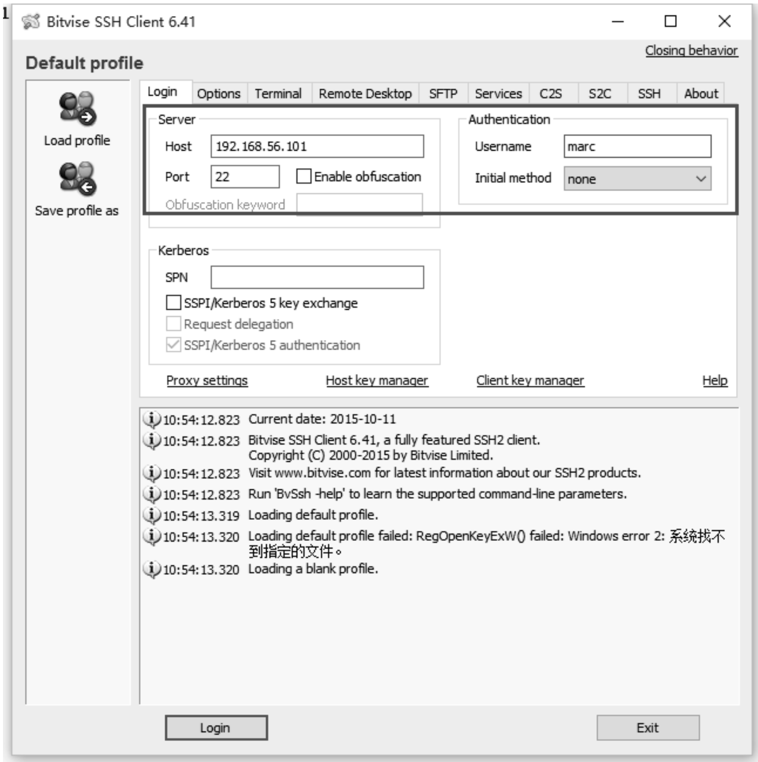


图2-15　Bitvise SSH Client

第一次登录后，会提示接受并保存主机密钥（host key），如图2-16所示。

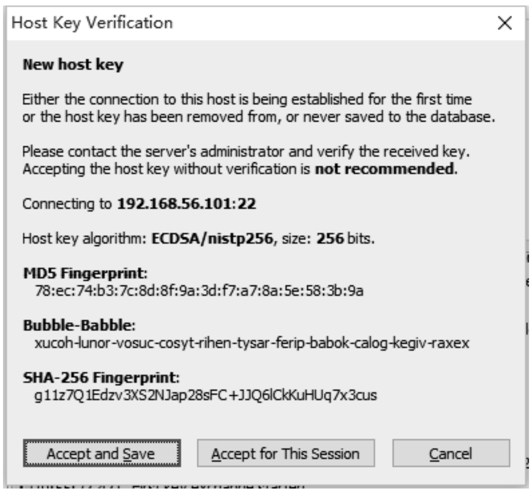


图2-16　提示接受并保存主机密钥

按照提示输入密码，如图2-17所示。

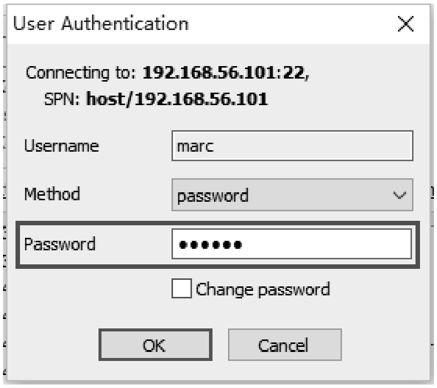


图2-17　输入密码

点击OK按钮即可登录到虚拟机，如图2-18所示。

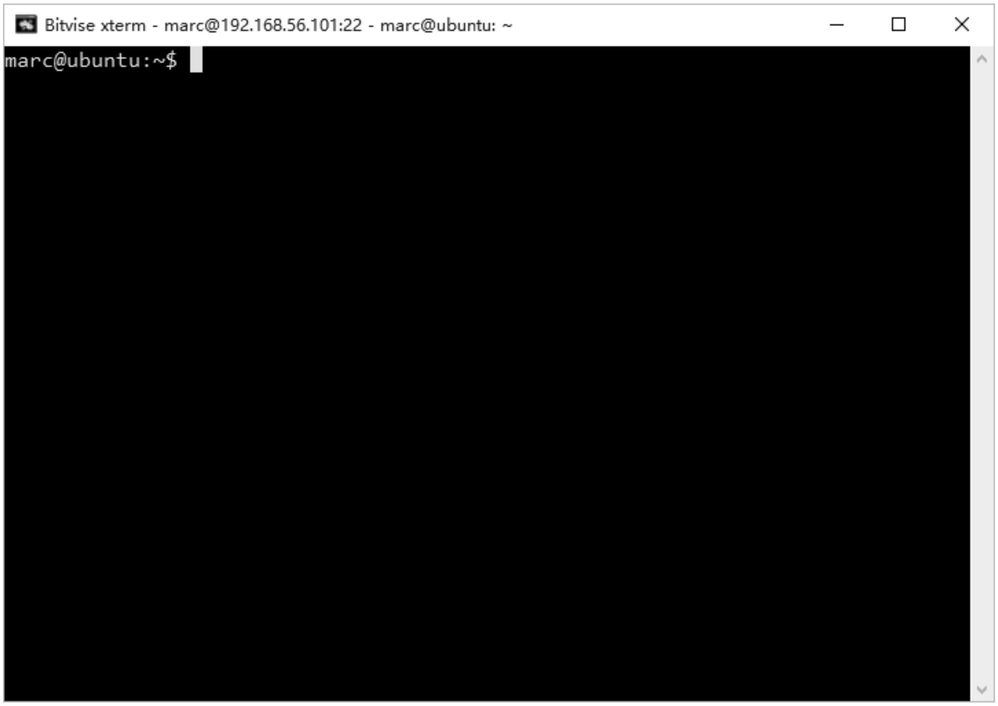


图2-18　登录到虚拟机上

至此，虚拟机配置完毕！

2.1.2 安装 JDK

Spark的运行依赖JVM。Spark使用Scala语言开发，而Scala是一种像Java一样运行在JVM上的语言，完美地融合了面向对象编程与函数式编程。关于Scala语言的相关内容，可以参考本书后面的附录。

Spark 1.4.1要求的最低JDK版本是JDK 6，Hadoop从2015年4月发布的2.7.0开始，已经抛弃了JDK 6，只支持JDK 7+，所以我们建议选择JDK 7，尽可能同时支持Hadoop和Spark。取JDK 7最新的稳定版即可，优先64位版本。

JDK的官方下载地址为http://www.oracle.com/technetwork/java/javase/downloads/index.html，默认会提供最新版本的JDK下载，目前是JDK 8。早期版本的链接在底部，搜索关键词older可以定位到，如图2-19所示。

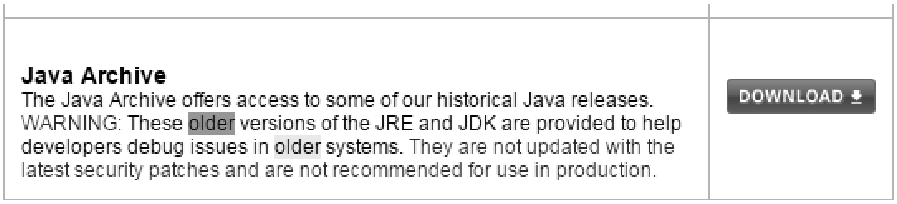


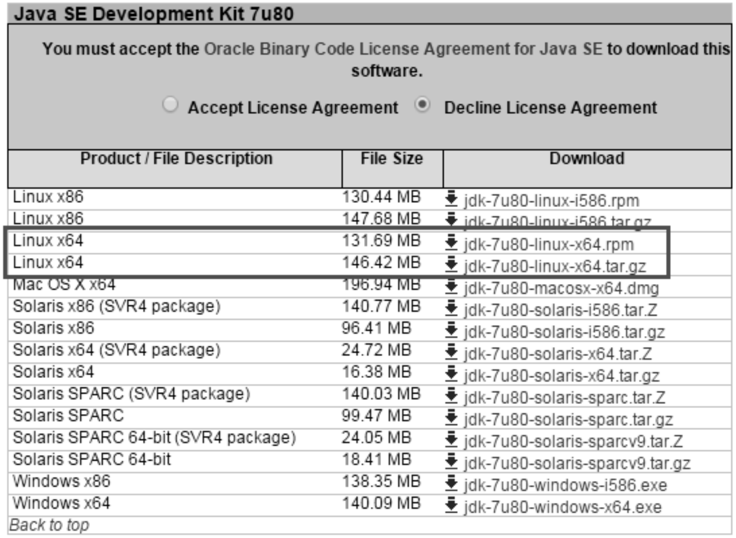
图2-19 早期版本

进入链接，发现有许多版本可供下载，这里我们选择Java SE 7，如图2-20所示。



图2-20 选择Java SE 7

进入该链接，会看到JDK 7的最新版本下载地址，其按平台又分很多种，这里选择Linux 64位的版本，如图2-21所示。



Java SE Development Kit 7u80

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

Accept License Agreement　　Decline License Agreement

Product / File Description	File Size	Download
Linux x86	130.44 MB	jdk-7u80-linux-i586.rpm
Linux x86	147.68 MB	jdk-7u80-linux-i586.tar.gz
Linux x64	131.69 MB	jdk-7u80-linux-x64.rpm
Linux x64	146.42 MB	jdk-7u80-linux-x64.tar.gz
Mac OS X x64	196.94 MB	jdk-7u80-macosx-x64.dmg
Solaris x86 (SVR4 package)	140.77 MB	jdk-7u80-solaris-i586.tar.Z
Solaris x86	96.41 MB	jdk-7u80-solaris-i586.tar.gz
Solaris x64 (SVR4 package)	24.72 MB	jdk-7u80-solaris-x64.tar.Z
Solaris x64	16.38 MB	jdk-7u80-solaris-x64.tar.gz
Solaris SPARC (SVR4 package)	140.03 MB	jdk-7u80-solaris-sparc.tar.Z
Solaris SPARC	99.47 MB	jdk-7u80-solaris-sparc.tar.gz
Solaris SPARC 64-bit (SVR4 package)	24.05 MB	jdk-7u80-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	18.41 MB	jdk-7u80-solaris-sparcv9.tar.gz
Windows x86	138.35 MB	jdk-7u80-windows-i586.exe
Windows x64	140.09 MB	jdk-7u80-windows-x64.exe

Back to top

图2-21　选择版本

有RPM包或tar.gz包对应不同的安装方法，RPM包安装之后，会替代之前安装的rpm包，作为系统默认的JDK，tar.gz包的灵活性则大很多，但稍复杂一些，这里只是体验，我们选择RPM包来安装。

安装RPM包的具体方法如下。

(1) 下载jdk-7u80-linux-x64.rpm到本地。

(2) 上传至虚拟机（使用SecureCRT时，可以用`rz -bye`命令，而Bitvise SSH Client提供了图形化的方式）。

(3) 在虚拟机上执行`sudo alien -i -c -v jdk-7u80-linux-x64.rpm`命令，其中`alien`相当于Ubuntu下的rpm，如果是其他Linux，安装命令可能是`rpm -ivh jdk-7u80-linux-x64.rpm`。默认情况下，JDK会安装到目录/usr/java/jdk1.7.0_65下。

(4) 设置环境变量`JAVA_HOME`。为了使下次登录还能生效，可以将命令追加至~/.profile末尾：`export JAVA_HOME=/usr/java/jdk1.7.0_65`。

tar.gz包的安装方法为：直接将文件复制至指定目标目录（比如/usr/java）下，解压缩即可，然后像上面的过程一样设置环境变量`JAVA_HOME`。

这两种安装方式的差异仅仅是目录的不同，安装后都需要配置环境变量`JAVA_HOME`。

执行`$JAVA_HOME/bin/java-version`，若能看到如下信息，则说明安装成功：

```
java version "1.7.0_65"
Java(TM) SE Runtime Environment (build 1.7.0_65-b17)
Java HotSpot(TM) 64-Bit Server VM (build 24.65-b04, mixed mode)
```

2.1.3　下载 Spark 预编译包

完成前面的准备工作后，我们开始下载Spark已经预编译的安装包，并运行示例。

1. 下载解压

Spark预编译包的下载地址为http://spark.apache.org/downloads.html，选择版本（这里使用的是1.4.1版本）、包类型（这里选择的是Pre-built for Hadoop 2.6 and later）之后，就可以开始下载了，如图2-22所示。

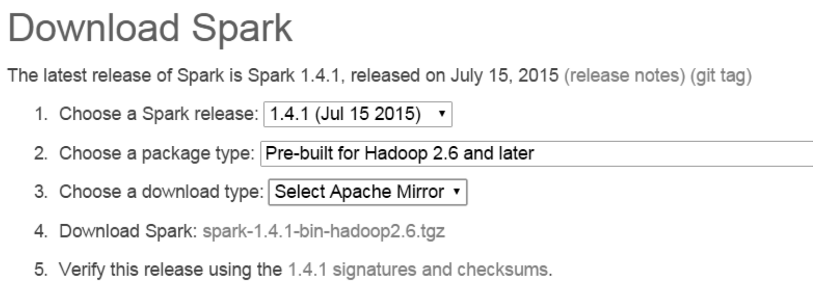


图2-22　下载Spark预编译包

下载到本地之后，为了确保下载的内容正确，最好做一下校验，此时用md5就好：

```
md5sum spark-1.4.1-bin-hadoop2.6.tgz
```

如果结果与下载链接上提供的结果一致，说明下载正确。

解压缩包，进入Spark目录：

```
tar xvf spark-1.4.1-bin-hadoop2.6.tgz
cd spark-1.4.1-bin-hadoop2.6
```

2. 运行示例

运行计算圆周率`Pi`的测试代码（日志忽略）：

```
./bin/run-example SparkPi 10 2>/dev/null
```

运行结果：

```
Pi is roughly 3.141276
```

Spark除了支持Scala语言，还支持Python语言。下面是运行Python版本的示例：

```
./bin/spark-submit examples/src/main/python/pi.py 10 2>/dev/null
```

运行结果：

```
Pi is roughly 3.138396
```

从Spark 1.4开始，Spark开始支持R语言（注：之前的SparkR只是R的一个包，目前已经停止更新）。

测试命令：

```
./bin/spark-submit examples/src/main/r/dataframe.R 2>/dev/null
```

运行结果：

```
Loading required package: methods
Attaching package: 'SparkR'
The following object is masked from 'package:stats':
    filter
The following objects are masked from 'package:base':
    intersect, sample, table
Warning message:
package 'SparkR' was built under R version 3.1.3
root
 |-- name: string (nullable = true)
 |-- age: double (nullable = true)
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)
    name
1 Justin
```

至此，Spark Local模式安装成功。

2.1.4 本地体验 Spark

Spark可以方便地进行交互式编程，这也是Scala、Python和R等语言的特性，下面以Scala语言为例进行介绍。执行下面的命令，进入Scala交互式编程模式：

```
./bin/spark-shell --master local[2]
```

其中--master用于指定Master服务器的地址，local表示运行在本地，[2]为可选项，表示启动两个工作线程。

启动后，我们在启动日志中可以看到这样的基本信息：

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.4.1
      /_/
Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_80)
```

同时，Master会启动一个HTTP Web服务，方便我们查看Spark的工作状态，默认绑定在本机所有IP的4040端口上。在浏览器中输入<LocalIP>:4040，得到的界面如图2-23所示。

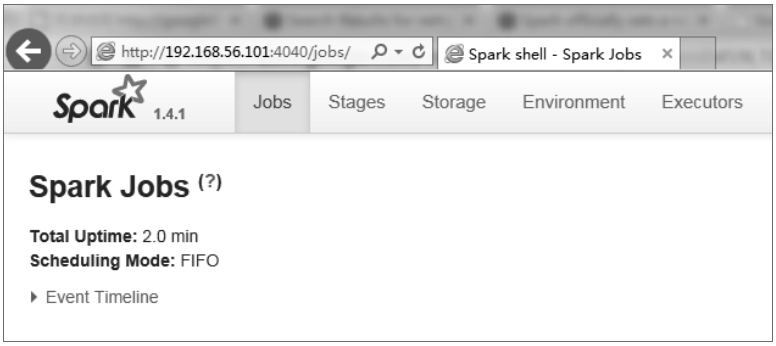


图2-23 Spark启动的Web界面

1. Hello, world

在交互式编程模式下，直接输入Scala的Hello world代码：

```
object HelloWorld {
    def main(args: Array[String]) {
        println("Hello, world!")
    }
}
HelloWorld.main(null)
```

执行结果如下：

```
scala> object HelloWorld {
     |   def main(args: Array[String]) {
     |     println("Hello, world!")
     |   }
     | }
defined module HelloWorld
scala> HelloWorld.main(null)
Hello, world!
```

在不涉及Spark包的情况下，这其实就是一个标准的Scala解释器。

Spark的核心数据结构是RDD，全称为弹性分布式数据集（Resilient Distributed Dataset）。最重要的是，它是一个数据集，所以我们把它想象成一个数组就好了，先不用管它的内部存储结构。

下面我们先从本地文本文件创建一个RDD：

```
scala> val textFile = sc.textFile("README.md")
textFile: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[3] at textFile at
<console>:21
```

RDD支持的方法不同于普通数据集，主要分为两类。

- 转换（Transformation）。生成新的RDD。
- 动作（Action）。返回一个非RDD类型的值，视方法而不同。

2. Action

先体验一下Action类型的操作：

```
scala> textFile.count() // 返回RDD元素的个数
res0: Long = 98 // sc.textFile创建的文本类型的RDD，每行一个元素，98表示文本的行数
```

执行完成后，检查一下Web界面，可以看到Job完成，并且被拆成一个Stage和两个Task，如图2-24所示。



图2-24　Spark Job

再来执行一下first操作：

```
scala> textFile.first() // 取RDD的第一个元素
res1: String = # Apache Spark // 第一行内容是# Apache Spark
```

执行完成后，Web界面就多了一条Job记录，按执行先后倒序排列，同样只有一个Stage，但Task数量是2，如图2-25所示。

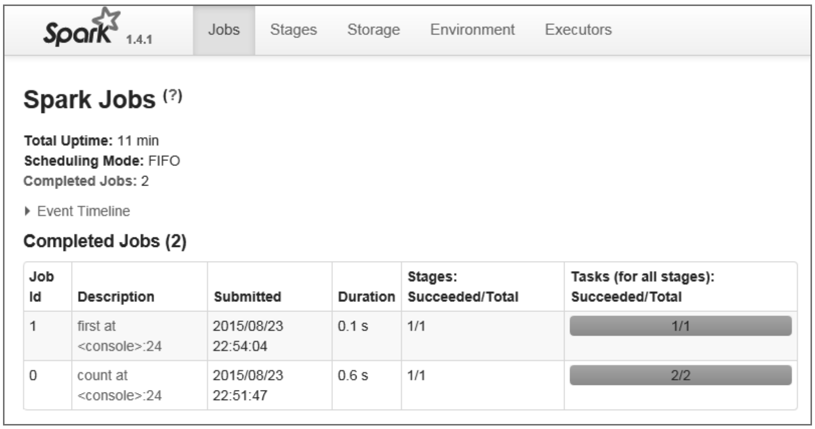


图2-25　Spark Job

图2-26是界面上的Stage信息。

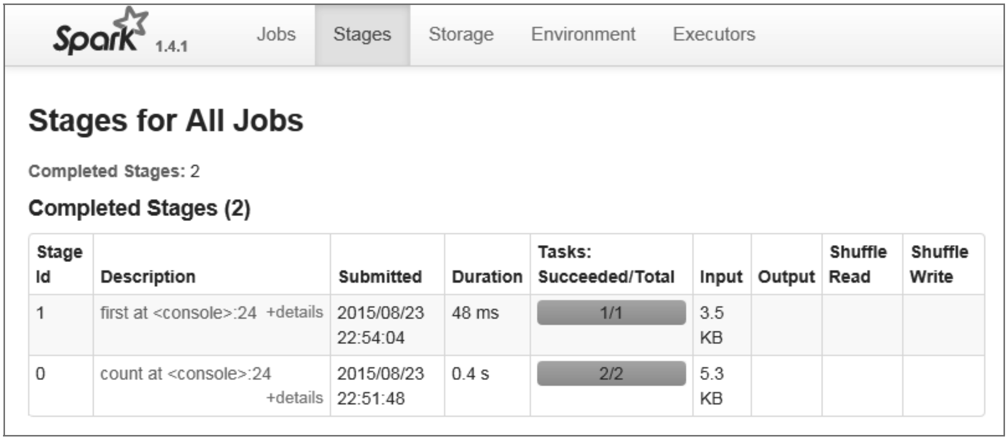


图2-26　界面上的Stage信息

3. 转换

再来体验一下转换操作，我们要统计README.md中包含关键字Spark的行数是多少：

```
scala> val linesWithSpark = textFile.filter(line => line.contains("Spark"))
linesWithSpark: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[2] at filter at
    <console>:23
scala> linesWithSpark.count()
res2: Long = 19
```

在上述代码中，`filter`转换接收一个函数作为输入，将每个RDD元素作为参数来调用这个函数，如果函数返回`True`，元素则保留在结果RDD中，否则抛弃。

这种非常简洁的定义函数的方法在Scala函数式编程中非常常见，常用的Scala语法可以参考本书后面的附录。

为了让代码更紧凑，更函数式一点，我们可以把两次调用合并起来写：

```
scala> textFile.filter(line => line.contains("Spark")).count()
res3: Long = 19
```

2.2　高可用 Spark 分布式集群部署

单机模式下，不但磁盘、内存、CPU资源有限，而且是单点，无法满足企业对可用性方面的高要求，因此实际部署时都会以集群模式部署。

本节会带领大家一步步部署一个满足线上需求的集群，而且具有高可用的特点，全集群没有单点，任何一台机器宕机时集群依然可以正常工作。

2.2.1　集群总览

整个集群除了核心的Spark之外，还有其他多个子集群，完整的集群结构如图2-27所示。为了实现集群的高可用性，我们引入了ZooKeeper集群进行主备切换。Spark没有存储能力，而且在生产环境下伴随大数据计算的一般都会有大数据存储，所以我们引入了Hadoop HDFS。同时，也引入了Hadoop YARN让集群具备非常好的扩展性，这主要是考虑到让Spark与传统Hadoop MapReduce并存，而且Hadoop生态中有许多非常优秀的系统也依赖Hadoop HDFS和YARN。

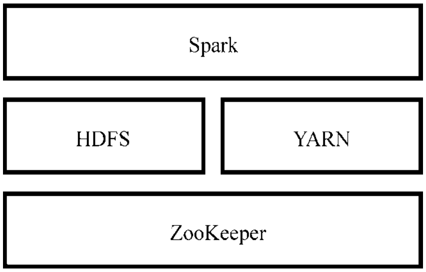


图2-27　ZooKeeper + Hadoop + Spark高可用集群结构

首先是一个ZooKeeper集群，集群的机器数量为2*n*+1台，可以抵御*n*台机器宕机的风险，其结构如图2-28所示。如果*n*取1，则总共3台，可以抵御1台机器宕机的风险。所有节点都是相同的角色，但运行中会自动选择一个作为leader。

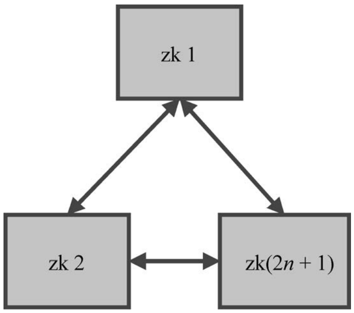


图2-28　ZooKeeper子集群的结构

ZooKeeper集群担负集群的终极仲裁者的角色，Spark集群的Master节点的主备切换通过与ZooKeeper集群协商完成，Hadoop集群中HDFS文件系统的NameNode节点以及YARN中的ResourceManager也都是通过ZooKeeper集群来协助完成主备切换。

其次是Spark集群，本书的主角，该集群有两类节点，其结构如图2-29所示。一类是Master节点，负责集群的资源管理。只要有一台集群就可以工作，但是一般至少两台，其中一台作为主节点，其他热备，当主Master节点异常下线时，其他节点通过ZooKeeper集群竞争选出新的主节点。另一类节点是Slave节点，用于执行计算任务，机器数量可以是一台，也可以是几十、几百

或上千台。国内互联网公司的最大集群已经达到上千台规模。

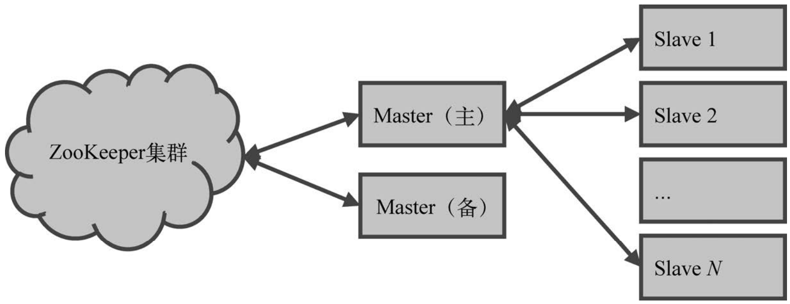


图2-29 Spark子集群的结构

此外，还有Hadoop集群。Hadoop从2.x开始，把存储和计算分离开来，形成两个相对独立的子集群：HDFS和YARN，MapReduce依附于YARN来运行。

Hadoop HDFS是一个分布式文件系统，为集群提供大文件的存储能力。NameNode管理所有的文件信息，一主一备，通过ZooKeeper来实现容灾切换，QJM节点的数量为$2n+1$，这里3台即可，负责记录NameNode的流水日志，确保数据不丢失，DataNode担负集群的存储负载，用于数据存储，如图2-30所示。

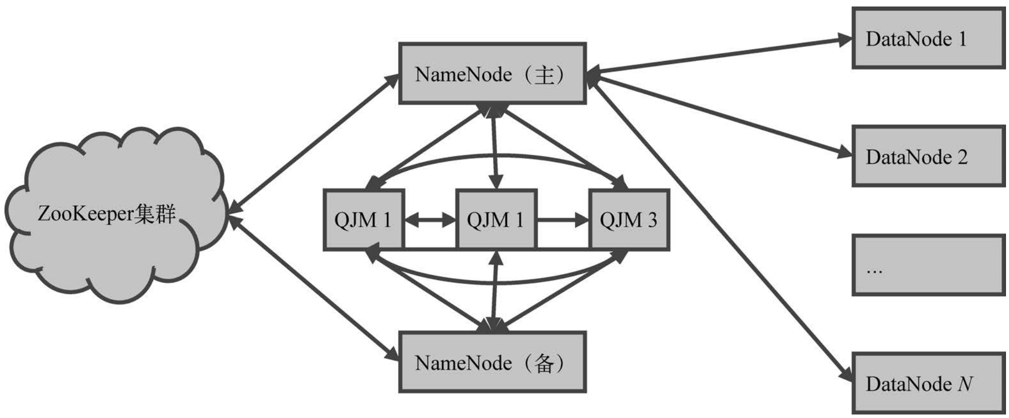


图2-30 HDFS集群结构

Hadoop YARN为MapReduce计算提供调度服务，而且也可以为其他符合YARN编程接口要求的集群提供调度服务，这也是Spark可以借助YARN来做资源调度的前提。该集群有两类节点：ResourceManager和NodeManager，其结构与Spark集群非常类似，其中ResourceManager为一主多备，NodeManager一般与DataNode部署在一起，实现最高效的数据访问，如图2-31所示。

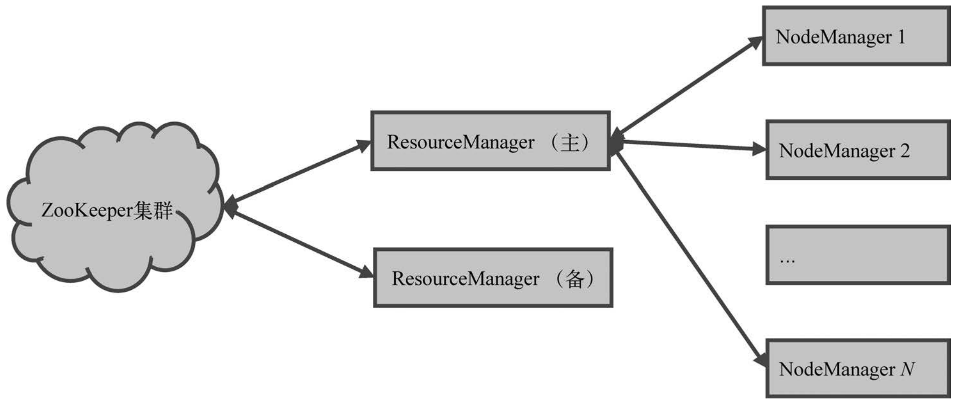


图2-31　YARN集群结构

2.2.2　集群机器的型号选择

Hadoop、Spark等集群的设计理念就是利用普通服务器协同工作来实现大数据的计算，因此集群可以部署在任何普通服务器上，甚至可以部署在个人笔记本电脑的多个虚拟机上。但集群各节点的工作性质不同，对硬件资源的消耗也不同，所有合理的搭配可以让资料得到更充分的利用。

我们从集群各类节点的工作性质上基本可以了解它们对于资源的消耗水平，具体如表2-1所示。

表2-1　集群各类节点的工作性质

集　　群	节　　点	主要工作	资源消耗
ZooKeeper	ZooKeeper	多个节点竞选leader 存储配置信息	不明显
Spark	Master	调度管理	少量内存
	Slave	任务计算	CPU、内存越多越好
Hadoop HDFS	NameNode	文件元数据存储、访问	内存越大支持的文件数量越多
	DataNode	存储数据	磁盘越大存的越多
	QJM	文件元数据日志存储	少量存储
Hadoop YARN	ResourceManager	调度管理	少量内存
	NodeManager	Hadoop MapReduce计算	CPU、内存 但Spark程序不需要

另外，考虑到无论是Spark计算还是Hadoop MapReduce计算，离数据越近移动数据的开销越小，而且在大数据计算的大部分场景下，移动数据的开销都会大于计算的开销，所以计算节点与存储节点一般都会混合部署。

因此，我们建议采用如下搭配。

- 这3类节点部署在相同的节点上：Spark Slave、HDFS DataNode、YARN NodeManager。而且这3类节点在集群中的数量最为庞大，最消耗资源，所有CPU、硬盘和内存资源都可以向它们倾斜。
- 其他节点的主要工作任务是管理。除了NameNode对内存要求高一点之外，其他对硬件消耗都不大，所以都可以混合部署。机器资源充足的可以考虑单独部署，毕竟节点数量都不多，按单节点容灾设计，最多也只消耗12台，最少3台就够。但这些节点中任何一个宕机，对集群都有不小的影响，宕机两台的话集群有可能就停止工作了，所以可以选择稳定性较高的配置，比如双电源、磁盘RAID。

具体到单机的硬件配置，结合Spark官方和我的实际经验，有如下建议。

- **存储**。Spark Slave节点尽可能靠近存储系统（比如HDFS和HBase），如果不能部署在相同节点上，那么请尽可能让它们在物理位置上更近一些以减少网络IO的开销，比如尽可能在相同机架上、同一个路由器下、相同机房里。而且每个节点建议使用4块或更多独立挂载的硬盘，并且不要配置RAID，这样可以提升Spark读写本地缓存数据时的效率。
- **内存**。Spark节点可以适应几百吉字节大小的内存。更多的内存可以缓存更多的数据，在一些迭代计算场景下性能提升效果非常明显。另外，建议分配最多节点内存总量的75%给Spark使用，其他的留给操作系统。实际使用多少，可以参考Web界面上的数据。
- **网络**。一些需要进行shuffle的byKey类操作，比如`groupByKey`、`reduceByKey`、`join`，它们的瓶颈一般都在网络上，所以建议使用10 Gbit/s及上网络。在Spark程序的Web界面上，可以看到shuffle对于网络使用的状况。
- **CPU**。Spark程序可以适应几十个内核的CPU，而且其计算性能与CPU核数量基本成正比。一般建议8~16个核，可以结合实际负载与成本综合考虑。一般情况下，如果数据已经在内存中了，CPU和网络IO会成为瓶颈。

总之，单机硬件配置会严重影响集群的性能，可以结合实际负载情况在成本与性能之间取得平衡。

2.2.3 初始化集群机器环境

机器配置好后，在正式部署前，需要初始化每台机器的基础环境，这里主要完成下面这3件事情：

- 创建账号；
- 安装JDK；
- 设置时间同步。

1. 远程批量操作工具fabric

上面的操作需要在所有机器上操作，既要传文件，还要执行一些命令。如果集群规模达到上

百台，手工登录的方式想想都恐怖，所以为了提升效果，最好能借助远程操作工具来进行，这里推荐Python下的fabric。

在Ubuntu下安装fabric的方法为：

```
sudo apt-get install fabric
```

其他平台下的安装方法请参考官方网站：http://www.fabfile.org/installing.html。

只需要一些基本的fabric用法，就可以帮助我们实现上百台机器的快速远程操作。这里以复制一个文件到3台机器上，查找某关键字为例来介绍。先编辑一个名为job.py的Python fabric配置文件：

```
#!/usr/bin/env python
# -*- encoding:utf-8 -*-
"""
fabric测试
"""

from fabric.api import *

def cp_and_grep(key):
    """
    复制文件至远程机器上，并查找通过参数`key`指定的关键词。
    """
    # 拼接shell登录
    command = "grep %s test.txt" % (key)
    # 在本地先执行一次
    local(command)
    # 传输本地文件test.txt至远程机器的/tmp目录下
    put('test.txt', '/tmp')
    # 设置远程机器的当前目录为/tmp
    with cd( '/tmp/' ):
    # 在远程机器上执行命令
    run(command)
```

调用方法如下：

```
fab \
    --fabfile=job.py \
    --hosts=<IP1>:<端口1>,<IP2>:<端口2>,<IP3>:<端口3> \
    --user=<ssh用户名> \
    --password=<ssh密码> \
    cp_and_grep:key="Spark"
```

下面简要介绍一下上述方法的参数。

- **--fabfile**。指定配置Python脚本文件。
- **--hosts**。指定远程机器IP、端口，多台机器之间使用逗号（,）分隔。
- **--user**。指定SSH登录时使用的账号，不指定时使用本地当前使用的系统账号。

- **--password**。指定SSH密码。
- **cp_and_grep**。它为要执行的Python配置脚本中的函数，这里需要设置参数key的值为"Spark"。

当机器数量太多的时候，在参数列表中列出全部机器略有些烦琐，而且这样的调用方式会让密码出现在shell历史命令中。我们可以在Python配置文件中指定机器列表、用户名以及密码，这通过在job.py中的cp_and_grep函数定义前添加如下代码来实现：


```
env.user = 'ssh用户名'
env.password = 'ssh密码'
env.hosts = ['<IP1>:<端口1>', '<IP2>:<端口2>', '<IP3>:<端口3>']
```

这样调用命令可以简化为：

```
fab \
    --fabfile=job.py \
    cp_and_grep:key="Spark"
```

而且虽然配置文件中指定了机器列表，在调试时还是可以手工指定目标机器：

```
fab \
    --fabfile=job.py \
    cp_and_grep:hosts="<IP1>:<端口1>;<IP2>:<端口2>",key="Spark"
```

在这3种指定机器的方式中，命令行中函数名后直接指定hosts（第三种方法）的优先级最高，通过env.hosts（第二种方法）其次，--hosts选项（第一种方法）最低。

默认情况下，fabric会按顺序逐台机器执行，我们可以使用--parallel选项让所有操作并行起来：

```
fab --parallel \
    --fabfile=job.py \
    cp_and_grep:key="Spark"
```

将配置文件的文件名改为fabfile.py，还可以省去选项--fabfile，调用命令简化为：

```
fab --parallel cp_and_grep:key="Spark"
```

此外，使用fab -l命令，可以查看fabfile.py中所有可用的函数列表：

```
$ fab -l

fabric测试

Available commands:

    cp_and_grep     复制文件至远程机器上，并查找通过参数key指定的关键词
```

2. 创建账号

有了fabric的帮助，给上百台机器添加账号也只是弹指一挥间的功夫。出于安全考虑，一般

不用root账号来运行集群。

fabric配置文件如下：

```
#!/usr/bin/env python
# -*- encoding:utf-8 -*-
from fabric.api import *
env.user = 'ssh用户名'
env.password = 'ssh密码'
env.hosts = ['<IP1>:<端口1>', '<IP2>:<端口2>', '<IP3>:<端口3>']
def add_user(user='spark'):
    """
```

接着创建用户，用户名通过参数`user`指定：

```
"""
run("""
user="%s"
if [ `grep "^$user:" /etc/passwd | wc -l` -eq 0 ]; then
    useradd --shell /bin/bash -m $user
fi
"""
% (user)
)
```

然后调用如下方法：

```
fab --parallel add_user:user=`spark`
```

此外，我们还可以创建一个公共的根目录，作为后续所有子系统安装的根目录，比如/data/ha_spark_cluster/。

3. 安装JDK

JDK的安装可以参考2.1.1节的说明，这里只是改造成批量安装。需要指出的是，考虑到部署的目标机器可能是新安装的操作系统，可能没有rpm或alien包，而且一般没有外网可以方便地安装这些工具，所以推荐使用tar.gz包来安装。此时只需要使用tar工具，该工具Linux一般都会自带。

fabric中的JDK安装函数如下：

```
def install_jdk_tar(tarfile, install_dir):
    """
```

其中参数`tarfile`指定JDK tar.gz包的全路径，`install_dir`指定安装路径：

```
"""
with cd( install_dir ):
    # 上传文件
    put(tarfile, '.')
    # tar xf, create link
    run("""
        pkg=`basename %s` # 包名，比如jdk-7u80-linux-x64.tar.gz
```

```
    tar xf $pkg # 解压包
    rm -f $pkg # 解压之后删除包
    jdkdir=`tar tf $pkg | head -1 | cut -d '/' -f1` # 获取解压之后的路径，比如jdk1.7.0_80
                                                    # 设置环境变量JAVA_HOME
    echo "export JAVA_HOME=`pwd`/$jdkdir" >> /etc/profile
""" % (tarfile)
)
```


4. 设置时间同步

集群要求所有节点的时间保持同步，并且误差一般要求不超过30秒，所以需要让每台机器定期调整一下操作系统的时间。

同步时间的命令如下：

```
/usr/sbin/ntpdate <NTP SERVER> ...
```

一般情况下，同步指定多台NTP Server，可确保高可靠性。

2.2.4　部署 ZooKeeper 集群

在所有子系统中，ZooKeeper的部署最简单，其节点数必须是奇数。每个节点的部署过程类似，大概分为如下几步。

(1) 复制包到所有节点机器的目标目录（比如/data/ha_spark_cluster/zookeeper），用 `scp` 命令或者 fabric 进行批量操作。注意设置目标目录的权限。

(2) 编辑配置文件并将其复制到所有节点。

安装包下的conf/zoo_sample.cfg为配置模板，其中有各配置项的详细说明：

```
# 服务器之间或客户端与服务器之间维持心跳（tick）的时间间隔，单位是毫秒
tickTime=2000

# 集群中的follower服务器与leader服务器之间，在初始连接时能容忍的最多心跳数
initLimit=10

# 集群中的follower服务器与leader服务器之间，在请求和应答之间能容忍的最多心跳数
syncLimit=5

# ZooKeeper保存数据的目录
# 不要用/tmp目录，因为机器重启之后会被删除，这里只是示例
dataDir=/tmp/zookeeper

# ZooKeeper监听的端口，客户端通过这个端口连接到ZooKeeper服务器
clientPort=2181

# 最大客户端连接数。客户端过多时可以增加本配置项的值
#maxClientCnxns=60
```

大部分配置都可以采用默认值，需要重新设置的是`dataDir`。默认情况下，它在/tmp目录下，重启之后数据可能丢失，需要将其设置为其他目录，比如/data/ha_spark_cluster/zookeeper/dataDir。

另外，我们需要在配置文件最后追加集群节点信息。下面以3个节点为例进行介绍：

```
server.1=<host1>:2888:3888
server.2=<host2>:2888:3888
server.3=<host2>:2888:3888
```

其中2888和3888代表两个端口号（可以换成其他端口），用于ZooKeeper节点之间的通信，最典型的场景是follower节点通过前面的端口连接leader，使用后面的端口来竞选新的leader。

需要注意的是，新的配置文件需要复制到所有ZooKeeper节点上去。

(3) 启动所有节点。

在每个节点上进入安装目录，执行如下命令：

```
./bin/zkServer.sh start
```

然后查看当前节点的状态：

```
$ ./binServer.sh status
JMX enabled by default
Using config: /usr/local/rt_cluster/zookeeper-3.4.6/bin/../conf/zoo.cfg
Mode: follower
```

可以发现，`Mode`的值为`follower`，说明当前节点不是leader节点。

(4) 连接测试。

在任意节点上执行./bin/zkCli.sh可以连接到当前节点的ZooKeeper服务器，进入交互式操作模式。

使用`help`命令可以查看帮助信息，具体如下：

```
[zk: localhost:2888(CONNECTED) 0] help
ZooKeeper -server host:port cmd args
    connect host:port
    get path [watch]
    ls path [watch]
    set path data [version]
    rmr path
    delquota [-n|-b] path
    quit
    printwatches on|off
    create [-s] [-e] path data acl
    stat path [watch]
    close
```

```
ls2 path [watch]
history
listquota path
setAcl path acl
getAcl path
sync path
redo cmdno
addauth scheme auth
delete path [version]
setquota -n|-b val path
```


2.2.5 编译 Spark

绝大部分情况下，我们不需要自己编译Spark，官方已经提供了非常多的匹配不同Hadoop版本的Spark包供下载，如图2-32所示。

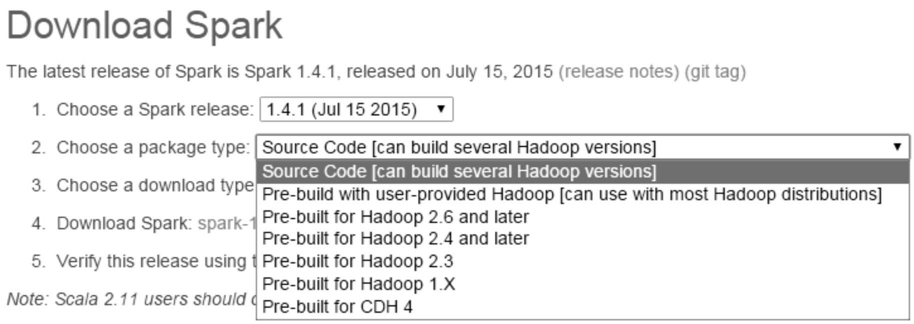


图2-32 选择Spark包

除了Hadoop发行版本不同外，默认的安装包包括如下特性：

- 支持Java 6+；
- 使用Scala 2.10；
- 支持JDBC服务和命令行客户端、集成Hive 0.13.1（目前也只能集成这个版本），支持Hive和Hive Thrift server。

如果这些已经能够满足你的需要了，可以跳过本节，直接选择一个合适的预编译版本下载下来就好了。

如果你有下面这些特定需求，可以考虑重新编译Spark：

- 指定特定的HDFS版本；
- 指定特定的YARN版本，可以与HDFS不一样，仅支持2.2.0及更高版本；
- 使用Scala 2.11。

Spark的运行版本使用mvn来编译，已经集成在源码中了。如果当前机器有外网，或者配置了http代理，就可以直接调用编译命令来进行编译了。

编译命令如下：

```
./build/mvn \
    -Phadoop-x.y \
    -Dhadoop.version=x.y.z \
    -Pyarn \
    -Dyarn.version=x.y.z \
    -Phive -Phive-thriftserver \
    -Dscala-2.11 \
    -DskipTests clean package
```

下面简要介绍各个参数的含义。

- **`-Phadoop-x.y`**。用于指定Hadoop的主版本号，默认的选项是`-Phadoop-2.2`。
- **`-Dhadoop.version=x.y.z`**。可选项，用于指定HDFS子版本号。比如，对于Hadoop 1.*x*系列，可以这样：

  ```
  -Phadoop-1 -Dhadoop.version=1.2.1
  ```

  对于Cloudera的CDH“mr1”系列，则可以这样：

  ```
  -Phadoop-1 -Dhadoop.version=2.0.0-mr1-cdh4.2.0
  ```

- **`-Pyarn`**。用于开启YARN功能。
- **`-Dyarn.version=x.y.z`**。可选项，YARN默认使用与HDFS相当的版本，如果不同，则可以用此选项指定。比如在下面的示例中，HDFS版本是2.3.0，YARN版本是2.2.0：

  ```
  -Phadoop-2.3 -Dhadoop.version=2.3.0 -Pyarn -Dyarn.version=2.2.0
  ```

- **`-Phive -Phive-thriftserver`**。用于开启JDBC和Hive 0.13.1功能。
- **`-Dscala-2.11`**。用于指定Scala版本为2.11，而不是默认的2.10。
- **`-DskipTests`**。这里忽略测试过程。
- **`clean package`**。`clean`和`package`是编译目标，`clean`执行一些清理工作，比如删除上一次打包时的痕迹，`package`用于编译并打包。

另外，在编译之前，可以设置让mvn使用更大的内存以提升效率：

```
export MAVEN_OPTS="-Xmx2g -XX:MaxPermSize=512M -XX:ReservedCodeCacheSize=512m"
```

最后，我们还可以在编译的同时指定打包成.tgz的形式，就像官网上提供的下载包一样，方便发布，具体方法是使用`./make-distribution.sh --name custom-spark --tgz`来替换上面的编译命令`./build/mvn`。

下面是一个示例，用于指定Hadoop版本为最新的2.7.1，开启YARN和JDBS，默认使用的是Scala版本，编译后打包成.tgz的形式：

```
./make-distribution.sh \
    --name spark-1.4.1-bin-hadoop2.7.1 --tgz \
    -Phadoop-2.7 -Dhadoop.version=2.7.1 \
    -Pyarn \
    -Phive -Phive-thriftserver \
    -DskipTests clean package
```

2.2.6 部署 Spark Standalone 集群

部署Spark Standalone集群的过程主要分为3步。

(1) 在每个 Master 节点上，配置运行账户 ssh 免密码登录到每台 Slave 节点上。假设我们的集群包含 2 个 Master 节点，10 个 Slave 节点，那么需要配置 $2 \times 10=20$ 次。

此外，出于安全考虑，建议不要使用root账户来运行Spark，所以最好在每台机器上创建一个专用账户。

要配置从A节点用x账户ssh免密码登录到B节点，首先在A机器上切换到x用户下，执行如下命令：

```
ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa
```

接着将执行上述命令后生成的文件~/.ssh/id_dsa.pub的内容，追加至B机器x用户下的文件~/.ssh/authorized_keys。

若配置好，那么在A机器上用x用户执行`ssh B`命令时不需要输入密码。

为何一上来就要搞这么麻烦的配置呢？因为后续的一大堆集群管理脚本都依赖这一特性，而且我们在分发JDK、安装包、配置文件时也会受益。另外，Hadoop部署时，也需要配置ssh免密码登录。

理论上，可以不做这一项配置，只需设置一个环境变量`SPARK_SSH_FOREGROUND`，然后每次启停集群时，每个Slave节点都需要输入数次密码，如果节点达到上百个，则需要输入数百次密码，相当烦琐。

(2) 把 Spark 压缩包复制到每个机器上，并解压缩。

(3) 编写配置文件，分发到所有节点。这里有两个配置文件是必需的，具体如下。

- conf/slaves。里面是所有Slave节点的地址，以IP或`hostname`的形式标识，每行一个。比如，有3个Slave节点，它们通过IP来标识，则文件内容是：

```
192.168.1.3
192.168.1.4
192.168.1.5
```

- conf/spark-env.sh。里面是Spark集群的一些环境变量，比如前面提到的`JAVA_HOME`也在这里定义。

安装包提供了模板spark-env.sh.template，我们在它的基础上编辑即可：

```
cp spark-env.sh.template spark-env.sh
```

spark-env.sh是一个标准的bash脚本，里面有很多注释，可以帮助我们理解配置项。所有配置项都非必须，只是为了满足更个性化的部署需求。

这里我们配置一下`JAVA_HOME`即可（如果部署时配置了系统环境变量，这里也可以省略）。

配置文件准备好后，将其分发到所有节点上。

前面部署的集群虽然可以运行，但有一个风险：如果任何一台机器宕机，则集群就失效了，这对于企业级应用来说是不能接受的。

下面我们来介绍如何打造具备高可用（简称HA）特性的集群：集群在任意一台机器宕机的情况下，依然可以正常运转。如果单机的可靠性是99%的话，那么容许任意一台机器宕机的集群的可用性可以提高至99.99%，效果非常显著。

在Standalone模式下，集群可以容许某一个或多个工作节点失效。当工作节点失效时，Master会自动将未完成的任务转移至其他节点重新计算，但Master节点本身是一个单点，是HA的瓶颈。

Spark提供了两种级别的HA解决方案，具体如下所示。

- 第一种方案是使用ZooKeeper，多个Master节点都连接到这个ZooKeeper集群，通过ZooKeeper竞选出一个活动的Master节点，其他节点热备。当活动Master节点死亡之后，其他的Master节点再次竞选出一个活动节点，重新竞选恢复服务的过程只需要1~2分钟，这期间只有新任务受影响不能提交，老任务可以继续执行。

  显然，这种方案非常适合线上正式环境，满足任何一个节点宕机集群都可以继续工作的高可用要求，但需要额外部署一套ZooKeeper集群，好在ZooKeeper对硬件要求不高，可以与Master节点或Hadoop Namenode节点共用机器资源。

  前面我们已经部署好了ZooKeeper集群，Spark配置使用ZooKeeper，只需要在conf/spark-env.sh中设置环境变量`SPARK_DAEMON_JAVA_OPTS`，并在它的内容中指定以下3个参数。

  - **spark.deploy.recoveryMode**。设置为`ZOOKEEPER`。
  - **spark.deploy.zookeeper.url**。ZooKeeper集群地址（比如，192.168.1.100:2181和192.168.1.101:2181）。
  - **spark.deploy.zookeeper.dir**。在ZooKeeper上的目录（比如default:/spark）。

  示例代码如下：

```
export SPARK_DAEMON_JAVA_OPTS="-Dspark.deploy.recoveryMode=ZOOKEEPER
-Dspark.deploy.zookeeper.url=192.168.1.1:2181,192.168.1.2:2181,192.168.1.3:2181
-Dspark.deploy.zookeeper.dir=/spark"
```

- 第二种方案是针对单个节点的。如果你不需要ZooKeeper那么高的可用性，只是希望Master进程退出之后，再重启时可以恢复到之前的状态，那么Spark提供了另外的选择：文件系统模式。在这种模式下，Master会在本地磁盘中写入足够的信息，在重启时可以确保之前中断的任务可以继续完成。

  虽然我们不推荐使用这种方式，但还会介绍一下其配置方法：设置SPARK_DAEMON_JAVA_OPTS参数的值，涉及的选项名称如下所示。

  - **spark.deploy.recoveryMode**。设置为FILESYSTEM。
  - **spark.deploy.recoveryDirectory**。本地目录，用于保存恢复信息。

  示例代码如下：

  ```
  export SPARK_DAEMON_JAVA_OPTS="-Dspark.deploy.recoveryMode
      =FILESYSTEM -Dspark.deploy.recoveryDirectory
      =/data/spark/recoveryDirectory/"
  ```

  部署完成后，就可以启动集群了，具体方法是登录到一个Master节点，进入安装目录，执行如下命令：

  ```
  ./sbin/start-all.sh
  ```

  此时会启动本机作为Master，并且启动conf/slaves中的所有节点作为Slave。

其他的Master节点也需要单独启动Master服务：

```
./sbin/start-master.sh
```

多个Master会自动连接ZooKeeper，竞选出一个活跃的Master，其他的进入热备状态。

停止集群的方法是：

```
sbin/stop-all.sh
```

此外，sbin目录下还提供了更多细粒度的集群操作工具，具体如下。

- **sbin/start-master.sh**。在本机启动一个Master实例。
- **sbin/stop-master.sh**。停止本机通过bin/start-master.sh启动的Master实例。
- **sbin/start-slave.sh**。在本机启动一个Slave实例。
- **sbin/stop-slave.sh**。停止本机通过bin/start-slave.sh启动的Slave实例。
- **sbin/start-slaves.sh**。在conf/slaves指定的每台机器上启动一个Slave实例。
- **sbin/stop-slaves.sh**。停止所有通过sbin/start-slaves.sh启动的Slave实例。

启动之后，可以通过当前活跃的Master的Web界面http://<active-master-ip>:8080查看集群的运行状况。

2.2.7　高可用Hadoop集群

Spark的优势是计算，不提供存储功能，而且本身的资源调度策略比较简单，所以与Hadoop混合部署，可以充分利用HDFS的存储功能以及YARN的调度优势。这种混合部署是一种最常用的部署方式。

Hadoop从2.*x*时代开始，内部拆分成了独立的几个模块，存储与计算功能相对独立。

- HDFS为分布式文件系统，专门负责文件存储。
- MapReduce（简称MR）是计算框架，提供计算支持。
- YARN是资源调度模块，MapReduce程序由YARN来调度运行，同时YARN还可以为其他系统提供调度服务，比如Spark。

Spark可以单独使用HDFS或YARN，或者两者同时使用。

Hadoop的部署不在这里详述，这里只是给出一些部署建议。

在Hadoop 2.*x*之前的版本中，NameNode节点是单点，一旦NameNode节点宕机，整个集群将无法工作，只能手工恢复。Hadoop 2.4之前的YARN，它的ResourceManager也是单点，直到2.4才支持双机热备自动容灾切换，所以出于可用性考虑，建议部署2.4及以上版本的Hadoop。

对于已经部署但没有配置HA的Hadoop系统，增加配置HA相对于完整部署简单多了，只需要更新一些配置，再增加部署$2n + 1$个QJM节点或者使用可选的网络文件系统（NFS）即可，具体可参考官方文档或其他资源。

2.2.8　让Spark运行在YARN上

在Spark Standalone模式下，集群资源调度由Master节点负责。Spark也可以将资源调度交给YARN来负责，其好处是YARN支持动态资源调度。Standalone模式只支持简单的固定资源分配策略，每个任务固定数量的core，各Job按顺序依次分配资源，资源不够时排队等待。这种策略适用单用户的场景，但在多用户时，各用户的程序差别很大，这种简单粗暴的策略很可能导致有些用户总是分配不到资源，而YARN的动态资源分配策略可以很好地解决这个问题。关于资源调度，第3章中还会详细讲解。

另外，YARN作为通用的资源调度平台，除了为Spark提供调度服务外，还可以为其他子系统（比如Hadoop MapReduce、Hive）提供调度，这样由YARN来统一为集群上的所有计算负载分配资源，可以避免资源分配的混乱无序。

在Spark Standalone集群部署完成之后，配置Spark支持YARN就相对容易多了，只需要进行如下两步操作。

(1) 在conf/spark-env.sh中增加一项配置`HADOOP_CONF_DIR`，指向Hadoop集群的配置文件目录，比如：

```
export HADOOP_CONF_DIR=/usr/local/rt_cluster/hadoop-2.6.0/conf
```

这里配置同样需要分发至所有节点。

(2) 重启集群。

另外，即便不部署Hadoop集群，Spark程序还是可以访问HDFS文件的：添加一些依赖的jar文件，然后通过以hdfs://开头的完整路径即可。但缺点也很明显，因为HDFS与Spark节点是分离的，数据移动成本很高，大部分情况下都会大于计算成本，因此应用的局限性很明显，不适合大数据量时的计算。

经过上述的部署，Spark可以很方便地访问HDFS上的文件，而且Spark程序在计算时，也会让计算尽可能地在数据所在的节点上进行，节省移动数据导致的网络IO开销。

Spark程序由Master还是YARN来调度执行，是由Spark程序在提交时决定的。以计算圆周率Pi的示例程序为例，Spark程序的提交方式是：

```
$ ./bin/spark-submit --class org.apache.spark.examples.SparkPi \
    --master spark://<active-master-ip>:<port> \
    lib/spark-examples*.jar \
    10
```

其中参数`--master`决定调度方式：如果该参数的值以`spark://`开头，则使用Spark自己的Master节点来调度；如果其值是`yarn-client`或`yarn-cluster`，则是使用YARN来调度，而YARN的具体地址会从前面配置的Hadoop配置目录下的配置文件中得到。

YARN调度有如下两种模式。

- **yarn-cluster模式**。YARN会先在集群的某个节点上为Spark程序启动一个称作Master的进程，然后Driver程序会运行在这个Master进程内部，由这个Master进程来启动Driver程序，客户端完成提交的步骤后就可以退出，不需要等待Spark程序运行结束。这是一种非常适合生产环境的运行方式。
- **yarn-client模式**。跟yarn-cluster模式类似，这也有一个Master进程，但Driver程序不会运行在Master进程内部，而是运行在本地，只是通过Master来申请资源，直至程序运行结束。这种模式非常适合需要交互的计算。

Spark程序在运行时，大部分计算负载由集群提供，但Driver程序本身也会有一些计算负载。在yarn-cluster模式下，Driver进程在集群中的某个节点上运行，基本不占用本地资源。而在yarn-client模式下，Driver会对本地资源造成一些压力，但优势是Spark程序在运行过程中可以进行交互。所以，建议只在有交互需求的情况下才使用yarn-client方式，其他都使用yarn-cluster模式。

下面还是以计算圆周率为例来说明，因为不需要本地交互，所有可以使用yarn-cluster模式来运行：

```
$ ./bin/spark-submit --class org.apache.spark.examples.SparkPi \
    --master yarn-cluster \
    lib/spark-examples*.jar \
    10
```

2.2.9　一键部署高可用 Hadoop + Spark 集群

前面的部署过程需要我们一步步地手工操作，费时费力，而且一旦出错，一般都得从头再来。对于一个Spark新人来说，整体过程可能需要一周的时间。为了让新人们可以快速体验Spark集群的威力，我整理了一些脚本，汇总之后形成了一套脚本工具，在这套工具的协助下，只需要下载一些安装包，做一下简单的配置，即可完成上述高可用Hadoop + Spark集群的部署。如果机器及网络资源到位，1~2小时即可完成部署。该工具位于GitHub：https://github.com/marc-chen/hadoop-spark-installer。

1. 准备机器

所有机器分为两类：Master和Slave。

Master机器的要求如下。

- 安装软件：sshd、Python 2、fabric。
- 3台即可，建议硬件、软件配置完全相同。
- 对硬件性能要求不高，一般服务器都可以满足。

Slave机器的要求如下。

- 启动sshd服务。
- 建议3~100台，硬件、软件配置完全相同，存储、CPU、内存越大越好，具体可参考Hadoop和Spark对硬件的要求。

2. 快速安装

具体的安装步骤如下。

(1) 下载安装包。
(2) 编辑配置文件 conf/hosts 和 conf/config，具体可以参考目录下的模板文件。
(3) 手工下载需要的项目安装包到 packages 目录，包括 ZooKeeper、Hadoop、Spark，选择最新的稳定版本即可。
(4) 执行一键安装：

```
./install.sh all
```

注意事项
- 请确保直接使用root登录执行安装，不要使用su，否则fabric可能出现不兼容的情况。
- 如果密钥登录不起作用，尝试重启sshd。

3. 启动服务

进入Master机器的安装目录（比如/usr/local/myhadoop/），执行admin.sh会看到帮助信息：

```
> ./admin.sh
Usage: ./admin.sh {zookeeper|hadoop|spark} {start|stop}
```

先启动ZooKeeper：

```
./admin.sh zookeeper start
```

再启动Hadoop，如果是第一次启动，需要先初始化一下，相关命令为：

```
cd hadoop
./namenode_format.sh
cd -
```

接着启动Hadoop：

```
./admin.sh hadoop start
```

最后启动Spark：

```
./admin.sh spark start
```

通过这样简单的几步操作，即可部署完集群。

2.3　Spark 编程指南

Spark用Scala语言开发，而且Scala语言的函数式编程特性让代码的可读性非常高，所以本节主要以Scala语言为示例来讲解运行在Spark上的程序的编写。对于第一次接触Scala语言的读者，可以参考附录快速了解Scala的语法。

2.3.1　交互式编程

Scala本身是一种非常灵活的语言，既可以实现大型的系统（比如Spark），也可以当作脚本语言来使用，而且支持交互式编程。

Spark在设计之初，就把编程的易用性作为一项核心目标来实现。此外，Spark程序也支持交互式编程，这个特性无论是对于新人还是经验丰富的程序员来说，都带来非常棒的便利性。

下面我们先来看一下Hadoop MapReduce程序的过程。即使是官方文档2.7.1版本中提供的最简单的WordCount示例程序，也至少需要下面这几步。

(1) 用文本编辑器编写 61 行 Java 代码，并将其保存在一个文件中。

(2) 编译代码。

(3) 打包成 JAR 包。

(4) 提交至 Hadoop 集群执行。

相比之下，Spark提供了交互式编程，直接交互式输入代码，就可以看到代码马上被执行并显示执行结果。下面以WordCount为例，执行下面的命令进入交互式编程环境：

```
./bin/spark-shell
```

屏幕滚动上百行日志之后，我们看到了与普通Scala解释器一样的交互式界面。在日志中可以看到这样的信息：

```
Spark context available as sc.
```

这说明解释器已经帮助我们初始化了一个`SparkContext`对象的实体，名为`sc`，我们可以直接使用。

输入下面的代码，就会自动编译、提交并马上运行：

```
val textFile = sc.textFile("README.md")
val wordCounts = textFile.flatMap(line => line.split(" ")).map(word
    => (word, 1)).reduceByKey((a, b) => a + b)
wordCounts.collect()
```

执行完成后，得到的结果如下：

```
res1: Array[(String, Int)] = Array((package,1), (this,1), (Version"]
    (http://spark.apache.org/docs/latest/building-spark.html
    #specifying-the-hadoop-version),1), (Because,1), (Python,2),
    (cluster.,1), (its,1), ([run,1), (general,2), (have,1), (pre-built,1),
    (locally.,1), (locally,2), (changed,1), (sc.parallelize(1,1), (only,1),
    (several,1), (This,2), (basic,1), (Configuration,1), (learning,,1),
    (documentation,3), (YARN,1), (graph,1), (Hive,2), (first,1), (["Specifying,1),
    ("yarn-client",1), (page](http://spark.apache.org/documentation.html),1),
    ([params]`.,1), (application,1), ([project,2), (prefer,1), (SparkPi,2),
    (<http://spark.apache.org/>,1), (engine,1), (version,1), (file,1),
    (documentation,,1), (MASTER,1), (example,3), (distribution.,1),
    (are,1), (params,1), (scala>,1), (systems.,1...
```

2.3.2 RDD 创建

Spark计算都是围绕RDD进行的。首先，我们需要创建一个RDD，生成方式总结起来有两大类：第一类是由Driver程序的数据集生成，第二类是由外部数据集生成，比如共享文件系统、HDFS文件和HBase等。

1. 从Driver程序的数据集生成RDD

从Driver数据集生成RDD的最直接的方法是使用`SparkContext`对象的`parallelize`方法，其参数是一个`Seq`对象。由于Scala支持方便的隐式类型转换，因此其他可以被自动转换成Seq的

对象也可以作为parallelize的参数，比如Array。此外，Seq的子类对象也可以作为参数，比如List。下面是使用Seq作为输入参数的示例：

```
val data = Seq(1, 2, 3, 4, 5)
val rdd  = sc.parallelize(data)
```

或者可以从一个Array对象生成RDD：

```
val data = Array(1, 2, 3, 4, 5)
val rdd  = sc.parallelize(data)
```

或者从一个List对象生成RDD：

```
val data = List(1, 2, 3, 4, 5)
val rdd  = sc.parallelize(data)
```

parallelize还可以指定第二个可选参数：RDD的分区[partition，过去也称为切片（slice）]数量。Spark在运行时，一般RDD操作会为每个RDD分区运行一个Job。关于Job，最简单的理解可以是它对应一个Java线程。可以根据集群的情况指定初始分区的数量，如果不指定，系统也会默认设置一个。默认值由配置项spark.default.parallelism决定，本地模式下默认取CPU的核数量。

此外，SparkContext对象还提供了range方法，通过指定长整数的开始值、结束值以及步长，生成一定范围的长整型，最终调用parallelize生成RDD。

值得注意的是，因为这种方式的数据源在Driver程序中受到Driver所在节点的资源限制，不适合处理特别大的数据，尤其是接近或超过本机资源的数据，所以更多适用于交互式环境下的程序调试或数据量较小的数据。

2. 从外部数据集生成RDD

前面已经提供了从Driver数据集生成RDD的方式，但受制于Driver程序所在的单节点的资源能力，不能处理数据特别大的场景，比如在处理HDFS文件时，动辄100 TB的数据，远超过本地的资源能力，而且数据的移动成本也相当高。

所以，对于这种外部数据集，Spark提供了专门的方式生成RDD，即从分布式的数据直接生成分布式的RDD，这使得RDD从生成到计算的全过程都是分布式的，不会形成资源瓶颈。

Spark可以从任何Hadoop支持的存储类型的数据源生成RDD，包括本地文件系统、HDFS、Cassandra、HBase、Amazon S3等，Spark支持文本文件、Sequence文件，以及其他任何Hadoop InputFormat格式的数据。

文本文件可以使用SparkContext对象的textFile方法生成，其参数是文件的地址，可以是本地文件系统的完整路径，也可以是以hdfs://、s3n://等开头的URL，比如前面提到的WordCount示例从本地文件生成RDD：

```
scala> val textFile = sc.textFile("README.md")
textFile: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[1] at textFile at
<console>:21
```

此外，使用文本文件时，还有几点注意事项，具体如下。

- 如果文件是通过本地文件系统路径指定的，必须确保所有计算节点都可以通过这个路径访问到该文件，所以要么每个节点的该路径下都有一份副本，或者使用网络文件系统。
- Spark 所有基于文件的生成 RDD 的方法（除了 `textFile`，还有 `hadoopFile`、`sequenceFile`、`objectFile`、`binaryFiles`等），都支持目录、压缩文件和通配符。比如，我们可以这样调用：

```
textFile("/my/directory")
textFile("/my/directory/*.txt")
textFile("/my/directory/*.gz")
```

- 像`parallelize`一样，`textFile`也支持指定分区数量。默认情况下，Spark为每个数据块（比如HDFS的默认数据块大小是128 MB，早期使用过64 MB）创建一个分区，我们可以指定更多的分区数量，但不能减少。

除了最基本的文本文件外，Spark还支持其他类型的文件。

- wholeTextFiles可以一次读取一个目录下的许多小文件，并返回<文件名、内容>二元组，而不是像`textFile`那样每条记录是一行文本，不保留文件信息。
- 对于SequenceFile格式的文件（Hadoop针对大数据计算使用的一种文件格式），可以使用`SparkContext`对象的`sequenceFile[K, V]`方法，其中`K`和`V`分别代表文件中key和value的类型，这些类型必须继承自Hadoop的`Writable`接口，比如`IntWritable`和`Text`。不过Spark提供了便利，可以直接使用Scala基本类型，比如`Int`和`String`，它们已经实现了`Writable`接口，所以我们可以直接这样使用：`sequenceFile[Int, String]`。
- 对于其他Hadoop InputFormat，我们可以使用`SparkContext`对象的`hadoopRDD`方法，其定义如下：

```
def

hadoopRDD[K, V](conf: JobConf, inputFormatClass: Class[_ <:
    InputFormat[K, V]], keyClass: Class[K], valueClass: Class[V],
    minPartitions: Int = defaultMinPartitions): RDD[(K, V)]
```

  其中的参数跟Hadoop任务一样。对于新的Hadoop MapReduce API（`org.apache.hadoop.mapreduce`），请使用`newAPIHadoopRDD`方法。`SparkContext`中还有一个`hadoopFile`方法，是对`hadoopRDD`的封装，使用默认Hadoop配置。同样，`newAPIHadoopFile`是对`newAPIHadoopRDD`的封装。

- 对于HBase文件，因为HBase使用HDFS作为存储，也是一种Hadoop InputFormat，所以也是用`hadoopRDD`或`newAPIHadoopRDD`来读取数据的，比如：

```
val conf = HBaseConfiguration.create()
conf.set(TableInputFormat.INPUT_TABLE, "MyTable")
val hBaseRDD = sc.newAPIHadoopRDD(conf
    , classOf[TableInputFormat]
    , classOf[org.apache.hadoop.hbase.io.ImmutableBytesWritable]
    , classOf[org.apache.hadoop.hbase.client.Result]
)
```


注意　上面的代码不能直接运行，需要添加HBase相关的JAR包到classpath中并在代码中引用相关的包。

此外，Spark还提供了RDD对象的序列化与反序列化功能。通过RDD.saveAsObjectFile保存的RDD序列化对象，可以使用SparkContext.objectFile重新加载进来，这虽然不如Avro等其他序列化方式效率高，但却非常简单好用。

SparkContext对象还提供了binaryFiles方法，以二进制的形式直接读取Hadoop MapReduce计算的结果文件，比如我们有这些输出文件:

```
hdfs://a-hdfs-path/part-00000
hdfs://a-hdfs-path/part-00001
...
hdfs://a-hdfs-path/part-nnnnn
```

通过这样的调用：

```
val rdd = sparkContext.dataStreamFiles("hdfs://a-hdfs-path")
```

生成的RDD内容是：

```
(a-hdfs-path/part-00000, 文件内容)
(a-hdfs-path/part-00001, 文件内容)
...
(a-hdfs-path/part-nnnnn, 文件内容)
```

所有RDD创建方法，都在SparkContext对象中提供，完整的信息可以参考官方API文档。值得注意的是，从外部创建RDD不会马上读进内存中计算，而只是保存读取它们的信息，在计算时再根据需求就近读取，所以基本不用担心单点资源的瓶颈问题。

2.3.3　RDD 操作

RDD是Spark的核心抽象，所有计算都围绕RDD进行，生成RDD只是万里长征的第一步，然后我们可以对RDD进行各种操作，这些操作可以分为两类。

- Transformation（转换）。一个RDD经过计算后生成新的RDD，比如WordCount示例中的flatMap、map和reduceByKey。
- Action（动作）。返回结果到Driver程序中，这一般意味着RDD计算的结束，比如WordCount中的最后一步collect操作。

Spark下的所有Transformation都是一种lazy（懒洋洋）模式，就是说计算不会马上进行，而是先记录下计算方式，仅在有Action被触发需要向Driver程序返回结果时才会启动真正的计算。这种设计的好处是更高效，不需要将每次Transformation的非常大的结果返回给Driver程序，只需要将最终的一般会小很多的结果返回给Driver程序，而且过程中的数据也不需要有完整的快照保存，可以根据集群的资源使用情况动态调配。

默认情况下，RDD的Transformation实际计算只在有Action时才进行，而且RDD的结果都是临时的，被新的Transformation或Action使用过后即丢弃。但我们可以对RDD进行持久化或cache操作，这也会触发Transformation进行真正的计算，而且Spark会将RDD的结果保持在集群的内存或磁盘中，甚至复制多份，这样再次访问时不需要重复计算，就可以获取更快的响应速度。

1. 以WordCount为例理解执行过程

以下面的WordCount代码为例：

```
val textFile = sc.textFile("README.md")
val words = textFile.flatMap(line => line.split(" "))
val wordPairs = words.map(word => (word, 1))
val wordCounts = wordPairs.reduceByKey((a, b) => a + b)
wordCounts.collect()
```

第一行代码从本地文本文件生成一个RDD，每行一条记录，但并没有执行读文件操作。第二行代码将每行数据按空格拆分成单词，也没有马上执行。第三行代码将各个单词加上计数值1，没有马上执行。第四行代码对所有相同的单词进行聚合相加求各单词的总数，依然没有马上执行。最后一行代码中的`collect`是Action，需要返回结果到Driver程序，这才触发RDD开始实际的计算。首先，会触发`wordCounts`开始计算，而`wordCounts`依赖`wordPairs`，`wordPairs`依赖`words`，`words`依赖`textFile`。这其实是一条RDD依赖链条：`textFile`→`words`→`wordPairs`→`wordCounts`，然后从链条头部`textFile`开始依次计算，最终将结果返回给Driver程序。

当然，因为持续化操作也会触发RDD的计算，所以下面的调用也会真正的计算：

```
wordPairs.persist()
```

注意 `cache`方法等同于`persist`，但`persist`的表达更广泛，可以通过参数设置持久化到内存中或磁盘中。但无论哪种都是持久化，而`cache`只局限于持久化到内存中。

在交互式编程环境中，输入`WordCount`代码，Spark程序就会自动完成编译、打包、上传、执行的过程，并输出结果，这就是交互式编程的便利之处。此外，在交互式编程环境下，系统自动创建了一个`SparkContext`实体，名称是`sc`，我们可以直接使用它。

下面我们从RDD操作层面来逐行解释上面的代码：

```
val textFile = sc.textFile("README.md")
```

SparkContext对象的textFile函数接收一个文本文件作为输入，创建一个新的RDD，并返回给只读变量textFile：

```
textFile.flatMap(line => line.split(" "))
```

flatMap函数是一个RDD转换函数，接受一个函数作为输入，对当前RDD的所有成员调用输入的函数，并返回一个新的RDD。line => line.split(" ")是Scala函数式编程的简写形式，表示函数接收一个名为line的参数，函数体对line进行split操作，split的结果就是返回值。flatMap与map的区别是：map输入的是一个RDD成员，输出的也是一个RDD成员，RDD总成员数不变；而flatMap对每个输入的RDD成员，输出的是一个集合，集合里面的成员会被展开，这样输入一个输出多个。当然，也可以输出空集合。比如，当前行是“# Apache Spark”，按空格拆分后，返回一个数组，包含3个成员：“#”“Apache”和“Spark”，最终数组的3个成员会成为RDD的直接成员。

map函数接收一个函数作为参数，它会遍历RDD中的所有成员，并将成员作为参数来调用，输出的结果为新的RDD的成员：

```
map(word => (word, 1))
```

下面的代码按key聚合，数量相加：

```
reduceByKey((a, b) => a + b)
```

collect函数将RDD分布在各节点的数据拉回到Driver程序本地，交互式shell会自动显示内容。

```
wordCounts.collect()
```

Spark最核心的抽象是弹性分布式数据集（以下简称RDD），RDD可以通过HDFS文件创建，或者由其他RDD转换而来。我们先使用Spark目录下的README来创建一个新RDD：

```
scala> val textFile = sc.textFile("README.md")
textFile: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[1] at textFile at
<console>:21
```

2. 传递函数参数

Spark严重依赖传递函数类型的参数，比如常见的RDD Transformation函数map和reduce，接收的参数都是一个函数类型。

以Scala语言为例，有两种建议的方法，一种是匿名函数，适用于小片段的代码，前面WordCount示例中的函数参数都是这种类型。

另一种建议的方法是传递object对象中的静态方法（Scala object对象本身就是单实例的，它的方法都是静态方法），比如WordCount中的flatMap调用也可以这样调用：

```
object MyFunctions {
    def lineSplit(line: String): Array[String] = {
```

```
        line.split(" ")
    }
}
val words = textFile.flatMap(MyFunctions.lineSplit)
```

它等价于之前的匿名函数形式：

```
val words = textFile.flatMap(line => line.split(" "))
```

这是完整的WordCount示例：

```
object MyFunctions {
    def lineSplit(line: String): Array[String] = {
        line.split(" ")
    }
}
val textFile = sc.textFile("README.md")
val words = textFile.flatMap(MyFunctions.lineSplit)
val wordPairs = words.map(word => (word, 1))
val wordCounts = wordPairs.reduceByKey((a, b) => a + b)
wordCounts.collect()
```

当然，还有另外的方法（虽然不建议），那就是传入普通类的方法，但必须将方法所属的实例一起传进去，而不仅仅是方法本身。因为方法的运行依赖实例，方法的第一个隐含的参数this就指向实例。

下面还是以WordCount为例，这次我们来改写map调用。先定义map的参数函数：

```
import org.apache.spark.rdd.RDD
class MyClass extends java.io.Serializable {
    def wordMap(word: String): (String, Int) = { (word, 1) }
}
```

这里需要注意到，MyClass继承自java.io.Serializable，这让MyClass具备了序列化能力，为什么必须具备序列化能力呢？这得从RDD说起。前面提到过，RDD是弹性分布式数据集，实际计算是分布在各个节点上的，而RDD的计算过程都是在Driver程序中定义的，所以代码从Driver分发至各计算节点有一个过程，可以认为分为4步。

(1) 在 Driver 节点上序列化代码。
(2) 传送至各计算节点。
(3) 在计算节点上反序列化。
(4) 执行。

前面建议的两种方法（匿名函数和object对象方法）都具备序列化能力，而类默认是没有的，所以一个解决方法是让它添加一个父类java.io.Serializable。

然后可以这样来调用：

```
val myclass = new MyClass()
val wordPairs = words.map(myclass.wordMap)
```

这等同于：

```
val wordPairs = words.map(word => (word, 1))
```

知道了原理，就可以找出让`MyClass`不从`java.io.Serializable`继承也可行的办法，那就是使用类中的function替代method，因为function支持序列化：

```
class MyClass{
    val wordMap = (word: String) => (word, 1)
}
```

调用方法不变：

```
val myclass = new MyClass()
val wordPairs = words.map(myclass.wordMap)
```

3. 理解变量的作用域

Spark的核心是RDD，而RDD是分布式计算的。虽然Spark的设计可以让我们使用RDD时跟使用本地数据集非常类似，但仔细研究还是会发现一些不同之处，就像上面的案例那样，我们使用普通类的方法给RDD函数传递参数时，需要做一下特殊处理。

我们还是要稍微理解一点RDD的运作机制，以避免代码的运行结果与期望的不太一致。以下面的代码为例：

```
var counter = 0
val data = Seq(1, 2, 3)
data.foreach(x => counter += x)
println("Counter value: " + counter)
```

计数器`counter`的初始值为`0`，`foreach`遍历`data`中的所有元素，加到计数器上，最终可以看到如下运行结果：

```
Counter value: 6
```

如果我们将`data`转换成RDD，再来重新计算计数器：

```
var counter = 0
val data = Seq(1, 2, 3)
var rdd = sc.parallelize(data)
rdd.foreach(x => counter += x)
println("Counter value: " + counter)
```

会发现结果与对`data`直接`foreach`操作不太一样：

```
Counter value: 0
```

这是为什么呢？其实很简单，所有对RDD的函数调用，虽然代码上看起来是在Driver程序中运行的，但实际计算都不在本地。每个RDD操作都会被转换成Job分发至集群的执行器进程上运行，即便是单机本地运行模式，也是在单独的执行器进程上运行，与Driver程序隶属于不同的进程。每个Job的执行，都会经历序列化、网络传输、反序列化和运行的过程。

在上面的示例中，foreach中的匿名函数x => counter += x首先被序列化，然后传送至计算节点，反序列化之后再运行，因为foreach是Action操作，结果会返回到Driver进程中。

在序列化时，Spark会将Job运行所依赖的变量、方法（称为闭包）全部打包在一起序列化，相当于它们的一份副本。所以，在上面的示例中，counter会被一起序列化，然后传输于计算节点，反序列化生成一个新的counter变量，已经不是Driver程序中的变量了。在执行过程中，只是计算节点上的counter会自增，而Driver程序中的counter则不会发生变化。执行完成之后，结果返回到Driver程序中（示例中的foreach没有返回值），而Driver中的counter依然还是当初的那个Driver，值为0。

理解了RDD的过程，我相信读者就不难理解另外一个Spark编程时RDD操作的禁忌，那就是RDD操作不能嵌套调用，即在RDD操作传入的函数参数的函数体中，不可以出现RDD调用。

4. 理解以键值对为参数的操作

大部分Spark计算接收包含任意对象的RDD作为参数，少数计算要求RDD的元素必须是<key, value>对，最常用的是分布式的shuffle操作，比如按元素的key来分组或聚合。在Scala中，这些操作Tuple元组类型，<key, value>运算PairRDDFunction类，自动处理RDD元组。

比如，WordCount中的reduceByKey。

如果<key, value>中的key是用户自定义类型，注意必须实现了方法equals()以及hashCode()。

5. Transformation操作

Transformation操作是RDD两类操作之一，这些操作都接收一个RDD作为输入，返回一个新的RDD。除了以上示例给出来的操作，其他常用的操作说明如表2-2所示，详情请参考API文档。

表2-2　其他常用操作

Transformation操作	说　明
map(*func*)	对源RDD中的每个元素调用*func*，生成新的元素，这些新元素构成新的RDD并返回
flatMap(*func*)	与map类似，但每个输入的RDD成员可以产生0或多个输出成员（所以*func*的返回值是Seq类型）
filter(*func*)	对RDD进行过滤，对原RDD中的每个元素调用*func*，如果返回true，则保留该元素，由这些元素构成新的RDD并返回
mapPartitions(*func*)	与map类似，但map中的*func*作用的是RDD中的每个元素，而mapPartitions中的*func*作用的对象是RDD的一整个分区。所以*func*的类型是Iterator<T> => Iterator<U>，其中T是输入RDD元素的类型

（续）

Transformation操作	说　明
mapPartitionsWithIndex(*func*)	与mapPartitions类似，但输入会多提供一个整数表示分区的编号，所以*func*的类型是(Int, Iterator<T>) => Iterator<U>，多了一个Int
sample(*withReplacement*, *fraction*, *seed*)	对RDD进行抽样，其中参数*withReplacement*为true时表示抽样之后还放回，可以被多次抽样，false表示不放回；*fraction*为浮点数，表示抽样比例；*seed*为随机数种子，比如当前时间戳
union(*otherDataset*)	合并两个RDD，不去重，要求两个RDD中的元素类型一致
distinct([*numTasks*]))	对原RDD进行去重操作，返回的RDD中没有重复的成员
groupByKey([*numTasks*])	对<key,value>结构的RDD进行类似RMDB的group by聚合操作，具有相同key的RDD成员的value会被聚合在一起，返回的RDD的结构是(key, Iterable<value>) **注意：**groupByKey除了聚合，不对value进行任何操作，除非你调用完groupByKey之后没有进一步的操作，直接调用Action操作，否则建议使用下面的reduceByKey或aggregateByKey以获取更高的性能 **注意：**groupByKey涉及shuffle操作，默认输出的并发数量取决于输入RDD的分区数量，也可以通过可选参数[numTasks]手工指定
reduceByKey(*func*, [*numTasks*])	对<key, value>结构的RDD进行聚合，对具有相同key的value调用*func*来进行reduce操作，*func*的类型必须是(V, V) => V
sortByKey([*ascending*], [*numTasks*])	对<key, value>结构的RDD进行升序或降序排列
join(*otherDataset*, [*numTasks*])	对(K, V)和(K, W)进行join操作，返回(K, (V, W)) 外连接函数为leftOuterJoin、rightOuterJoin和fullOuterJoin

6. Action操作

Action操作是另外一类操作，Transformation操作结束之后，就该Action操作上场了，输出不再是RDD，也就意味着输出不是分布式了，而是回送至Driver程序。

表2-3是常用的Action操作列表，详情请参考API文档。

表2-3　常用的Action操作列表

Action操作	说　明
reduce(*func*)	对RDD成员使用*func*进行reduce操作，*func*接受两个参数，合并之后只返回一个值。reduce操作的返回结果只有一个值。需要注意的是，*func*会并发执行
collect()	将RDD读取至Driver程序，类型是Array，一般要求RDD不能太大
count()	返回RDD的成员数量
first()	返回RDD的第一个成员，等价于take(1)
take(*n*)	返回RDD前*n*个成员

（续）

Action操作	说　明
saveAsTextFile(*path*)	将RDD转换为文本内容并保存至路径path下，可能有多个文件。路径path可以是本地路径或者HDFS地址，转换方法是对RDD成员调用toString函数
saveAsSequenceFile(*path*)（仅Java和Scala）	与saveAsTextFile类似，但以SequenceFile格式保存，成员类型必须实现Writable接口或者可以被隐式转换为Writable类型（比如基本Scala类型Int、String等）
countByKey()	仅适用于(K, V)类型，对key计数，返回(K, Int)
foreach(*func*)	对RDD中的每个成员执行*func*，没有返回值，常用于更新计数器或输出数据至外部存储系统。这里需要注意变量的作用域

2.3.4　使用其他语言开发 Spark 程序

前面的例子都是基于Scala语言的，Spark也支持Python、Java和R。官方文档的示例程序也提供Scala、Python和Java版本，而本书的案例主要使用Scala。

2.4　打包和提交

bin/spark-shell只适合调试。正式环境下的Spark程序在调试好之后，需要编译、链接、打包成JAR包，然后提交至集群运行。

2.4.1　编译、链接、打包

以WordCount为例，首先将Spark程序代码保存到文件WordCount.scala中去：

```
/* WordCount.scala */
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
object WordCount {
    def main(args: Array[String]) {
        val conf = new SparkConf().setAppName("WordCount")
        val sc = new SparkContext(conf)
        val textFile = sc.textFile("README.md")
        val words = textFile.flatMap(line => line.split(" "))
        val wordPairs = words.map(word => (word, 1))
        val wordCounts = wordPairs.reduceByKey((a, b) => a + b)
        println("wordCounts: ")
        wordCounts.collect().foreach(println)
    }
}
```

用Scala写的Spark程序本身也是Scala程序，所以入口还是object中的main函数，建议文件名与object同名。

编译、链接、打包的工作由专用的构建工具来完成，一般使用sbt（simple build tool）或Maven。官方推荐用sbt，因为它更简单。

1. sbt安装

sbt需要手动安装，安装之前检查一下依赖项。

- ❑ sbt依赖Java，建议使用与Spark运行环境相同的JDK。
- ❑ sbt的安装和使用都需要使用外网，以便可以下载依赖的JAR包。如果没有外网，可以配置一下代理。在Linux下配置代理，只需设置一下环境变量`http_proxy`指向代理服务器即可。

先下载TGZ压缩包：

```
~$ wget 'https://dl.bintray.com/sbt/native-packages/sbt/0.13.9/sbt-0.13.9.tgz'
```

解压后进入目录，然后执行./bin/sbt，第一次执行时会从网上下载依赖的JAR包，并保存到~/.sbt：

```
~$ tar xf sbt-0.13.9.tgz
~$ cd sbt/
~/sbt$ ./bin/sbt
```


为了使用方便，可以加入到`PATH`路径中去：

```
~/sbt$ mv bin/ conf/ ~/
~/sbt$ export PATH="$PATH:$HOME/bin"
```

2. 编写sbt配置文件

一个典型的Spark程序的sbt配置文件如下，这里将其命名为wordCount.sbt：

```
name := "WordCount"
version := "1.0"
scalaVersion := "2.10.4"
libraryDependencies += "org.apache.spark" %% "spark-core" % "1.4.1"
```

wordCount.sbt本身也是Scala代码，如果使用了其他组件，可以在`libraryDependencies`中继续添加。

为了让sbt正常工作，sbt文件和程序源码需要满足一定的目录结构，应该是这个样子：

```
$ find .
.
./src
./src/main
./src/main/scala
./src/main/scala/WordCount.scala
./wordCount.sbt
```

3. 编译、链接、打包

相关命令如下：

```
$ bin/sbt package
...
[info] Packaging /data/wordCount/target/scala-2.10/wordcount_2.10-1.0.jar ...
[info] Done packaging.
[success] Total time: 37 s, completed Oct 11, 2015 11:33:13 AM
```

初次执行也比较慢，因为需要下载依赖的Spark相关的JAR包。

Spark程序除了依赖Spark，还依赖其他项目。此时需要创建一个assembly JAR包，其中包含所有递归依赖的JAR包，这可以借助sbt assembly插件来实现。

2.4.2　提交

打包之后，就可以提交至集群上运行了，提交任务的基本形式如下：

```
./bin/spark-submit \
    --class <main-class>
    --master <master-url> \
    --deploy-mode <deploy-mode> \
    --conf <key>=<value> \
    <application-jar> \
    [application-arguments]
```

比如，local模式下自带的计算Pi的示例程序这样提交：

```
$ ./bin/spark-submit \
>   --class org.apache.spark.examples.SparkPi \
>   --master local[2] \
> lib/spark-examples-1.4.1-hadoop2.6.0.jar \
> 10
```

其中`--class`参数指定包中的`object`对象。`--master`参数指定Spark集群地址，它可以是下面的任何一种。

- **`local[N]`**。表示本地模式。
- **`spark://host:port`**。表示Standalone模式。
- **`yarn`**。
- **`mesos://host:port`**。

`--deploy-mode`选项用于指定运行模式，可选的值为`client`或`cluster`，分别表示Driver程序是运行在本地还是运行在集群上，client模式仅适用于测试时的交互式运行，正式环境建议都使用cluster模式。

`spark-submit`提交程序时，会读取配置文件conf/spark-defaults.conf作为默认配置，命令行选项中的参数优先级高于配置文件，程序中的`SparkConf`对象的设置优先级最高。

此外，`--jars`选项可以用于指定额外的依赖包，集群上的所有节点都会访问这些文件。URL规则有3类，具体如下所示。

- 以file:/开头的本地文件，其他节点会通过Spark程序启动的http服务来下载。
- hdfs:、http:、https:、ftp，节点直接访问这些服务。
- 以local:/开头的文件，表示文件在本地，各节点直接在本地的该路径下访问，没有网络IO，可以提前手工传输到所有节点，或者通过网络文件系统，比如NFS。

`--help`选项用于查看所有支持的选项。

第 3 章

Spark工作机制

在熟悉了Spark的基本用法之后，本章将带领大家深入了解Spark的工作机制，主要包括调度管理、内存管理、容错机制、监控管理和配置管理。

3.1 调度管理

就像普通程序在运行时由操作系统来统一调度分配可以使用的CPU、内存资源一样，Spark集群上的程序要想申请使用资源，也要由统一的系统来调度分配，这就是调度管理系统。

如果集群上只有一个程序在运行，那么也没有必要进行调度了，如果多个程序在运行，就存在资源竞争的问题，所以调度管理最主要的目的是资源分配。

Spark集群上的资源主要是指CPU core数量和物理内存。core数量一般是物理CPU核的数量，我们也可以将其设置为CPU核数量的2倍或更多。在程序运行时，每个core对应一个线程。物理内存即所有机器节点分配给Spark集群的内存总量，一般每个节点最多会将80%的内存资源分配给Spark集群使用。

Spark自身支持资源调度，Standalone模式下由Spark集群中的Master节点来进行资源调度。此外，Spark还支持与其他调度管理系统一起工作，目前支持Hadoop YARN和Mesos，这样的好处有两个：一是外部调度系统可能支持更强大、灵活的调度策略；二是当Spark与其他分布式系统一起部署时，可以一起被统筹调度，避免Spark集群整体对资源的吞噬。

Spark调度按场景分为两类：一类是Spark程序之间的调度，这是最主要的调度场景；另外一类是Spark程序内部的调度，下面会详述。

3.1.1 集群概述及名词解释

为了更好地理解调度，我们先来鸟瞰一下集群模式下的Spark程序运行（参见图3-1）。

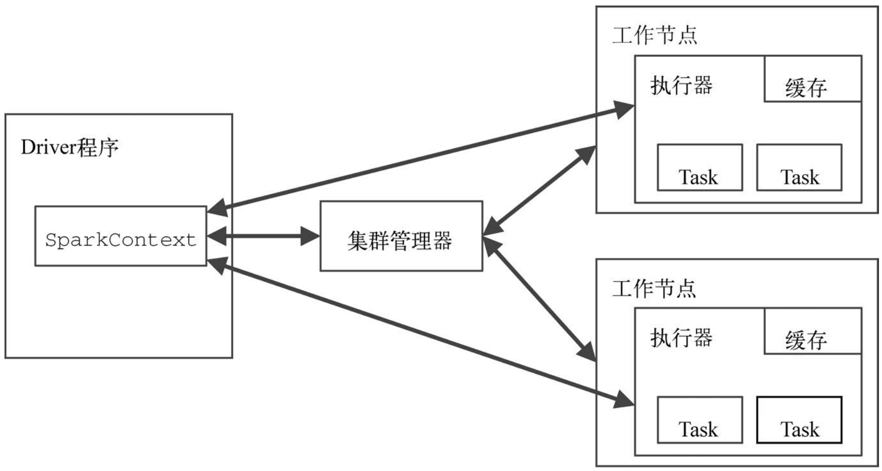


图3-1 集群模式下运行的Spark程序

1. Driver程序

在集群模式下，用户编写的Spark程序称为Driver程序。每个Driver程序包含一个代表集群环境的`SparkContext`对象并与之连接，程序的执行从Driver程序开始，中间过程会调用RDD操作（Transformation或Action），这些操作通过集群资源管理器来调度执行，一般在Worker节点上执行，所有操作执行结束后回到Driver程序中，在Driver程序中结束。

2. `SparkContext`对象

每个驱动程序里都有一个`SparkContext`对象，担负着与集群沟通的职责，大概过程如下。

(1) `SparkContext`对象联系集群管理器（cluster manager，视部署模式不同而不同），分配CPU、内存等资源。
(2) 集群管理器在工作节点（Worker Node）上启动一个执行器（专属于本驱动程序）。
(3) 程序代码会被分发到相应的工作节点上。
(4) `SparkContext`分发任务（Task）至各执行器执行。

3. 集群管理器

集群管理器负责集群的资源调度。Spark支持3种集群部署方式，每种部署对应一种资源管理器。

- Standalone模式（资源管理器是Master节点）。最简单的一种集群模式，不依赖于其他系统，调度策略相对单一，只支持先进先出（FIFO，First-In-First-Out）模式。
- Hadoop YARN（资源管理器是YARN集群）。它主要用来管理资源。YARN支持动态资源管理，更适合多用户场景下的集群管理，而且YARN可以同时调度Spark计算和Hadoop MR计算，还可以调度其他实现了YARN调度接口的集群计算，非常适用于多个集群同时部署的场景，是目前最主流的一种资源管理系统。

- **Apache Mesos（资源管理器是Mesos集群）**。Mesos是一个专门用于分布式系统资源管理的开源系统，与YARN类似，用C++开发，可以对集群中的资源做弹性管理。目前，Twitter、Apple等公司在使用Mesos管理集群资源。Mesos是高仿谷歌内部的资源管理系统Borg（论文已发表）实现的。

涉及的其他名词如下。

- **Allocation**。即“分配”。
- **App、Application、Spark程序**。泛指用户编写的运行在Spark上的程序，不仅是Scala语言编写的，其他支持的语言编写的也是。
- **节点、Worker节点**。集群上的计算节点，一般对应一台物理机器，仅在测试时会出现一台物理机器上启动多个节点。
- **Worker**。每个节点上会启动一个进程，名为Worker，负责管理本节点，运行jps命令可以看到Worker进程在运行
- **core**。Spark标识CPU资源的方式，对应一个或多个物理CPU核心，每个Task运行时至少需要一个core。
- **执行器**。每个Spark程序在每个节点上启动的一个进程，专属于一个Spark程序，与Spark程序有相同的生命周期，负责Spark在节点上启动的Task，管理内存和磁盘。如果一个节点上有多个Spark程序在运行，那么相应地就会启动多个执行器。
- **Job**。一次RDD Action对应一次Job，会提交至资源管理器调度执行。
- **Stage**。Job在执行过程中被分为多个阶段。介于Job与Task之间，是按Shuffle分隔的Task集合。
- **Task**。在执行器上执行的最小单元。比如RDD Transformation操作时对RDD内每个分区的计算都会对应一个Task。

3.1.2　Spark 程序之间的调度

Spark程序之间的调度资源分配策略有如下两种。

- **静态分配**。Spark程序启动时即一次性分配所有资源，运行过程中固定不变，直至程序退出。这是一种最简单可靠的分配策略，强烈建议优先使用这种策略，除非你非常确定这种方式无法满足需求。
- **动态分配**。运行过程中不断调整分配的资源，可以按需增加或减少。这比静态分配复杂得多，需要在实践中不断调试才能达到最优。

1. 静态资源分配

所有的集群管理器都支持静态资源分配，也就是说，每个Spark程序都分配一个最大可用的资源数量，而且在程序运行的整个过程中都持有它。这种方式在Spark Standalone模式、YARN和Mesos中都支持，资源分配可以进行如下配置（取决于不同集群）。

- Standalone模式。默认情况下，程序提交至Standalone模式的集群下时，会使用先进先出（FIFO）的顺序，而且每个程序都尝试尽可能使用所有可用的节点。我们可以限制程序使用的最大节点数量，通过设置`spark.cores.max`属性，或者通过设置`spark.deploy.defaultCores`来设置一个全局默认值，以防Spark程序没有设置。最后，除了控制core的最大数量外，每个Spark程序还可以设置`spark.executor.memory`来限制它能使用的最大内存数量。
- Mesos。为了使用静态分配策略，设置配置项`spark.mesos.coarse`的值为`true`，还可以设置`spark.cores.max`来限制每个Spark进程的最大core使用量，还需要设置`spark.executor.memory`来限制内存的使用。
- YARN。客户端`--num-executors`选项用于控制最多分配的执行器的数量，各执行器内部使用`--executor-memory`和`--executor-cores`来限制core的数量和内存大小。

Mesos上第二个选择是动态共享CPU核心数量。在这个模式下，每个Spark程序仍然有属于自己的固定且独立的内存资源（通过设置`spark.executor.memory`），但是当程序在某个机器上没有运行Task时，其他Spark程序会使用这些核心。这种模式非常适合那些数量非常多但不那么活跃的程序，比如来自不同用户的shell会话。然而，随之而来的风险是无法预估的延迟，因为当程序有Task需要执行时，从其他程序那里要回那些本属于自己的核心需要一些时间。要想使用这个模式，在使用mesos:// URL时不设置`spark.mesos.coarse`为`true`即可。

需要注意的是，目前所有模式下都没有在不同Spark程序之间提供内存共享的能力。如果你想用这种方式来共享数据，建议运行一个单独的服务程序来响应不同的请求去查询同一个RDD。在未来的发行版本中，in-memory存储系统（比如Tachyon）会提供另外的方式在不同的程序之间共享RDD。

2. 动态资源分配

Spark 1.2引入了基于负载情况的集群动态资源分配，分配给Spark程序的资源可以增加或减少。这意味着我们的程序可能会在不使用资源的时候将资源还给集群，需要的时候再从集群申请。这个特性在多个程序共享集群资源的时候特别有用。如果分配给Spark程序的资源中有一部分是空闲的，它就可以返还给集群放入资源池，这样可以被其他程序请求使用。在Spark中，动态资源分配的粒度是执行器，即增加或减少执行器。前面已经介绍过，Spark程序在机器的每个节点上只有一个执行器，所以增加或减少执行器意味着为Spark程序服务的节点数据量的增加或减少。通常，每台机器启动一个节点进程，但也不排除启动多个的情况发生。你可以通过设置`spark.dynamicAllocation.enabled`来启动这个功能。

这个特性当前只有在YARN模式下才可以使用，未来的版本也支持Standalone模式和Mesos下的coarse-grained模式。需要注意的是，虽然Mesos的fine-grained模式下也有一个类似的动态资源分配，但coarse-grained模式下的动态资源分配特性可以充分利用调度低延迟的优势。

- **配置动态资源分配**

这个特性的所有配置项都是类似spark.dynamicAllocation.*形式的命名。要想开启这个特性，

必须设置spark.dynamicAllocation.enabled的值为true，其他配置可以参考配置页。

此外，Spark程序还必须使用一个外部shuffle服务，这个服务的目的是保存执行器输出的shuffle文件，这样执行器可以安全移除。要想开启这个服务，请设置spark.shuffle.service.enabled的值为true。在YARN中，这个外部shuffle服务是集群中各个NodeManager的org.apache.spark.yarn.network.YarnShuffleService实现的。启动这个服务的步骤如下。

(1) 编译Spark时带上YARN选项。如果使用官网编译好的版本，可以忽略这一步。

(2) 找到spark--yarn-shuffle.jar，如果是自己编译Spark，这个文件应该在$SPARK_HOME/ network/yarn/target/scala-目录下；如果是官网编译好的版本，应该在lib目录下。

(3) 将这个jar包添加到集群中所有NodeManager的classpath中去。

(4) 在每个节点的yarn-site.xml文件中，在yarn.nodemanager.aux-services选项中添加spark_ shuffle，然后设置yarn.nodemanager.aux-services.spark_shuffle.class的值为org.apache.spark.network.yarn.YarnShuffleService。此外，设置所有的spark.shuffle. service.*选项。

(5) 启动集群中的所有NodeManager。

- **资源分配策略**

从最上面看，Spark应该在执行器不使用的时候扔掉它，然后在需要的时候再获取。由于没有准确的方式来预计一个即将被删除的执行器是否会在稍后马上运行一个Task，或者一个马上要添加的新执行器实际上是空闲的，因此我们需要一些启发式的方法来判断是否应该新增或删除执行器。

请求策略

开启动态资源分配之后，当Spark程序有Task处于排队待调度时，会请求额外的执行器资源，这种状况说明现有的执行器不足以让所有尚未执行完的Task的并发达到饱和。

Spark会一直循环请求执行器资源。实际请求的触发是在当排队Task已经等待了spark.dynamicAllocation.schedulerBacklogTimeout秒，然后只要还有排队的Task，就每隔spark.dynamicAllocation.schedulerBacklogTimeout秒请求一次。此外，每一轮请求的执行器的数量会成指数增长。比如，一个程序第一次会增加一个执行器，接下来的数量是2、4、8，以此类推。

指数增长策略的动机有两个：一是程序刚开始时申请执行器应该谨慎一些，因为万一只要一点点额外的执行器就够了，原理类似于TCP的慢启动；二是如果程序真的需要放多执行器，应该可以及时获取到。

移除策略

移除执行器的策略就相对简单多了，当Spark程序的某个执行器处于空闲状态的时间超过spark.dynamicAllocation.executorIdleTimeout秒之后会被移除。注意，在大部分情况下，这个条件与请求的条件是互斥的，即当有Task在等待执行时，此时是没有空闲执行器的。

● **执行器的优雅退出**

在出现动态分配之前，一个Spark执行器的退出要么是因为执行失败，要么是所属的Spark程序已经退出。不管哪种退出，执行器的所有附属状态信息都不再需要了，可以安全地直接丢弃。然而有了动态分配之后，执行器退出时进程可能还在运行。如果程序尝试去访问执行器中的数据，就会触发重新计算。因此，Spark需要一种在执行器优雅退出之前保存执行器状态的机制。

这个需求对于shuffle特别重要。在shuffle期间，Spark执行器首先将自己的map结果写入本地磁盘，然后作为一个服务来运行，其他执行器请求访问这些数据时进行响应。但是对于那些执行时间远超过同伴的Task来说，动态分配可能会在shuffle结束之前移除这个执行器，这样执行器写的shuffle文件就需要进行无谓的重新计算。

保存shuffle文件的方法是使用一个外部的shufle服务，这在Spark 1.2中已经引入了。这个服务指向一个长期运行的进程，在集群的每个节点上都会运行，而且独立于Spark程序和它们的执行器。如果这个服务被开启了，Spark执行器会从这些服务那里获取shuffle文件，而不是其他执行器。这意味着执行器写的任何shuffle状态在执行器退出之后还可以保留。

除了写shuffle文件外，执行器还会在磁盘或内存中缓存数据。当执行器被移除时，所有缓存的数据都不能被访问了。在Spark 1.2中还没有方法解决。在未来的发布中，缓存的数据可能会通过一个堆外的存储系统被保留下来，思路跟shuffle文件被一个外部的shuffle服务保存起来一样。

3.1.3 Spark 程序内部的调度

当Spark为多个用户同时提供服务时，我们可以考虑配置Spark程序内部的调度。

在Spark程序内部，不同线程提交的Job可以并行执行。Spark的调度器是线程安全的，因此可以支持这种需要同时处理多个请求的服务型应用。

默认情况下，Spark的调度器以FIFO的方式运行Job。每个Job分成多个Stage（比如map和reduce阶段），如果最前面Job的Stage有Task要运行，优先获取所有资源，然后才是第二个Job，以此类推。如果队列中的第一个Job不需要所有资源，那么第二个Job可以马上运行，但如果第一个Job特别大，则后面的Job都将会明显延迟。

从Spark 0.8开始，我们可以配置Job之间公平共享资源。在公平共享方式下，Spark采用“循环”（round robin）的方式为不同Job之间的Task分配资源，这样所有的Job可以获取差不多相同的资源。这就意味着，当有长时间的Job在运行时，短的Job在提交之后也可以马上运行并且取得不错的响应时间，而不用等长时间的Job结束。这种模式特别适用于多用户的场景。

要想开启程序内的公平调度，只需要在`SparkContext`中设置`spark.scheduler.mode`的值为`FAIR`：

```
val conf = new SparkConf().setMaster(...).setAppName(...)
conf.set("spark.scheduler.mode", "FAIR")
```

```
val sc = new SparkContext(conf)
```

1. 公平调度池

公平调度还支持对多个Job进行分组，这个分组称为调度池。每个调度池可以设置不同的调度选项，当我们想要为一些更重要的Job设置更高的优先级时，这个功能就非常有用了。再比如，我们可以为不同的用户设置不同的调度池，然后让各个资源池平等地共享资源，而不是按Job来共享资源。

不做设置的话，新提供的Job会自动进入默认调度池。我们可以指定让Job进入哪个调度池，具体方法是提交任务的线程在`SparkContext`中设置`spark.scheduler.pool`，如下所示：

```
sc.setLocalProperty("spark.scheduler.pool", "pool1")
```

这样设置之后，这个线程提交的所有Job（每个RDD Action都会触发提交一次Job，比如`count`、`collect`、`save`等）都会使用这个调度池。设置按线程来进行，这样可以很方便地让一个线程下的所有Job都在同一个用户下。如果要清空当前线程的调度池设置，可以这样设置：

```
sc.setLocalProperty("spark.scheduler.pool", null)
```

2. 调度池的默认行为

默认情况下，所有调度池平均共享集群的资源，默认调度池也是。但在每个调度池内部，各个Job是按FIFO的顺序来执行的。比如，你为每个用户创建一个调度池，那么每个用户平均共享集群资源，但每个用户的查询请求是按顺序进行的，而不是后面来的请求从这个用户前面的Job中挤占资源。

3. 调度池配置

调度池的配置也可以放在配置文件中，每个调度池有如下3个属性。

- **`schedulingMode`**。可以是FIFO或FAIR，用于控制调度池内的Job是排队执行还是平均共享资源。
- **`weight`**。用于控制调度池相对于其他调度池的权重，是一个相对值。默认情况下，所有调度池的权重是1，它们平均共享集群资源。如果某个调度池的权重是2，那么它就拥有权重为1的调度池2倍的资源。设置一个非常高的权重，比如1000，相当于配置了各个调度池的优先级。实际上，权重为1000的调度池只要有一个Job就会优先启动。
- **`minShare`**。除了权重，还可以设置一个最小资源值（core的数量）。公平调度器在按权限分配资源之前，总是会满足各个调度池的最小资源值。这样`minShare`可以保证调度池总是会获取一些资源，并且不被集群中其他高优先级的调度池抢走。`minShare`的默认值是`0`。

调度池的配置文件是一个XML文件，类似于conf/fairscheduler.xml.template。可以在`SparkConf`中指定配置文件：

```
conf.set("spark.scheduler.allocation.file", "/path/to/file")
```

XML文件的格式很简单，每个调度池对应一个`<pool>`节点，节点里面的成员就是调度池的属性，比如：

```
<?xml version="1.0"?>
<allocations>
    <pool name="production">
        <schedulingMode>FAIR</schedulingMode>
        <weight>1</weight>
        <minShare>2</minShare>
    </pool>
    <pool name="test">
        <schedulingMode>FIFO</schedulingMode>
        <weight>2</weight>
        <minShare>3</minShare>
    </pool>
</allocations>
```

完整的示例可以参考conf/fairscheduler.xml.template。配置文件中没有出现的调度池都被设置为默认值（`schedulingMode`为`FIFO`，`weight`为`1`，`minShare`为`0`）。

3.2 内存管理

相比Hadoop MapReduce来说，Spark计算具有巨大的性能优势，其中很大一部分原因是Spark对于内存的充分利用，以及提供的缓存机制。

3.2.1 RDD 持久化

持久化在早期被称作缓存（cache），但缓存一般指将内容放在内存中。虽然持久化操作在绝大部分情况下都是将RDD缓存在内存中，但一般都会在内存不够时用磁盘顶上去（比操作系统默认的磁盘交换性能高很多）。当然，也可以选择不使用内存，而是仅仅保存到磁盘中。所以，现在Spark使用持久化（persistence）这一更广泛的名称。

如果一个RDD不止一次被用到，那么就可以持久化它，这样可以大幅提升程序的性能，甚至达10倍以上。默认情况下，RDD只使用一次，用完即扔，再次使用时需要重新计算得到，而持久化操作避免了这里的重复计算，实际测试也显示持久化对性能提升明显，这也是Spark刚出现时被人称为内存计算的原因。

持久化的方法是调用`persist()`函数，除了持久化至内存中，还可以在`persist()`中指定storage level参数使用其他的类型，具体如表3-1所示。

表3-1 storage level参数

storage level	说　明
MEMORY_ONLY	默认的持久化级别，只持久到内存中（以原始对象的形式），需要时直接访问，不需要反序列化操作。内存不足时，多余的部分不会被持久化，访问时需要重新计算

（续）

storage level	说　明
MEMORY_AND_DISK	持久化到内存中，内存不足时用磁盘代替
MEMORY_ONLY_SER	类似于MEMORY_ONLY，但格式是序列化之后的数据（每个分区一字节数组），可以更节省内存，代价是消耗CPU
MEMORY_AND_DISK_SER	类似于MEMORY_ONLY_SER，内存不足时用磁盘代替
DISK_ONLY	只使用磁盘
*_2，比如MEMORY_ONLY_2和MEMORY_AND_DISK_2等	与上面的级别类似，但数据还复制到集群的另外一个节点上，总共两份副本，可提升可用性

此外，RDD.unpersist()方法可以删除持久化。

3.2.2　共享变量

Spark程序的大部分操作都是RDD操作，通过传入函数给RDD操作函数来计算。这些函数在不同的节点上并发执行，内部的变量有不同的作用域，不能相互访问，有些情况下不太方便，所以Spark提供了两类共享变量供编程使用——广播变量和计数器。

1. 广播变量

这是一个只读对象，在所有节点上都有一份缓存，创建方法是SparkContext.broadcast()，比如：

```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)
scala> broadcastVar.value
res0: Array[Int] = Array(1, 2, 3)
```

注意，广播变量是只读的，所以创建之后再更新它的值是没有意义的，一般用val修饰符来定义广播变量。

2. 计数器

计数器只能增加，可以用于计数或求和，默认是数值型，支持自定义类型。在Web界面上，也可以看到计数器共享变量。

计数器变量的创建方法是SparkContext.accumulator(v, name)，其中v是初始值，name是名称。注意，只有Driver程序可以读这个计算器变量，RDD操作中读取计数器变量是无意义的。

示例如下：

```
scala> val accum = sc.accumulator(0, "My Accumulator")
accum: org.apache.spark.Accumulator[Int] = 0
scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)
...
15/10/03 17:57:51 INFO DAGScheduler: Job 0 finished: foreach at <console>:24, took
0.649155 s
scala> accum.value
res1: Int = 10
```

3.3 容错机制

分布式系统通常在一个机器集群上运行，同时运行的几百台机器中某些出问题的概率大大增加，所以容错设计是分布式系统的一个重要能力。

3.3.1 容错体系概述

Spark以前的集群容错处理模型，像MapReduce，将计算转换为一个有向无环图（DAG）的任务集合，这样可以通过重复执行DAG里的一部分任务来完成容错恢复。但是由于主要的数据存储在分布式文件系统中，没有提供其他存储的概念，容错过程需要在网络上进行数据复制，从而增加了大量的消耗。所以，分布式编程中经常需要做检查点，即将某个时机的中间数据写到存储（通常是分布式文件系统）中。

RDD也是一个DAG，每一个RDD都会记住创建该数据集需要哪些操作，跟踪记录RDD的继承关系，这个关系在Spark里面叫lineage。由于创建RDD的操作是相对粗粒度的变换（如`map`、`filter`、`join`），即单一的操作应用于许多数据元素，而不需存储真正的数据，该技巧比通过网络复制数据更高效。当一个RDD的某个分区丢失时，RDD有足够的信息记录其如何通过其他RDD进行计算，且只需重新计算该分区，这是Spark的一个创新。

Spark的lineage也不是完美解决所有问题的，因为RDD之间的依赖分为两种，如图3-2所示。

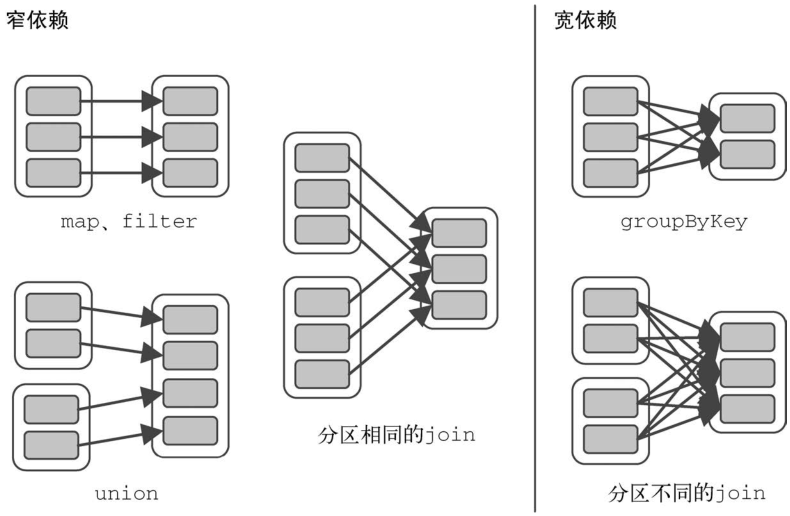


图3-2　RDD之间的两种依赖

根据父RDD分区是对应一个还是多个子RDD分区，依赖分为如下两种。

- **窄依赖**。父分区对应一个子分区。

- **宽依赖**。父分区对应多个子分区。

对于窄依赖，只需要通过重新计算丢失的那一块数据来恢复，容错成本较小。但如果是宽依赖，则当容错重算分区时，因为父分区数据只有一部分是需要重算子分区的，其余数据重算就造成了冗余计算。

所以，不同的应用有时候也需要在适当的时机设置数据检查点。由于RDD的只读特性使得它比常用的共享内存更容易做检查点，具体可以使用`doCheckPoint`方法。

在有些场景的应用中，容错会更复杂，比如计费服务等，要求零丢失。还有在Spark支持的Streaming计算的应用场景中，系统的上游不断产生数据，容错过程可能造成数据丢失。为了解决这些问题，Spark也提供了预写日志（也称作journal），先将数据写入支持容错的文件系统中，然后才对数据施加这个操作。

另外，Kafka和Flume这样的数据源，接收到的数据只在数据被预写到日志以后，接收器才会收到确认消息，已经缓存但还没有保存的数据在Driver程序重新启动之后由数据源从上一次确认点之后重新再发送一次。

这样，所有的数据要不从日志中恢复，要不由数据源重发，实现了零丢失。

3.3.2　Master 节点失效

Spark Master的容错分为两种情况：Standalone集群模式和单点模式。

Standalone集群模式下的Master容错是通过ZooKeeper来完成的，即有多个Master，一个角色是Active，其他的角色是Standby。当处于Active的Master异常时，需要重新选择新的Master，通过ZooKeeper的ElectLeader功能实现。关于ZooKeeper的实现，这里就不展开了，感兴趣的朋友可以参考Paxos。

要使用ZooKeeper模式，你需要在conf/spark-env.sh中为`SPARK_DAEMON_JAVA_OPTS`添加一些选项，详见表3-2。

表3-2　为`SPARK_DAEMON_JAVA_OPTS`添加的一些选项

系统属性	说　　明
`spark.deploy.recoveryMode`	默认值为`NONE`。设置为`ZOOKEEPER`后，可以在Active Master异常之后重新选择一个Active Master
`spark.deploy.zookeeper.url`	ZooKeeper集群地址（比如`192.168.1.100:2181,192.168.1.101:2181`）
`spark.deploy.zookeeper.dir`	用于恢复的ZooKeeper目录，默认值为`/spark`

设置`SPARK_DAEMON_JAVA_OPTS`的实际例子如下：

```
SPARK_DAEMON_JAVA_OPTS="$SPARK_DAEMON_JAVA_OPTS
    -Dspark.deploy.recoveryMode =ZOOKEEPER"
```

应用程序启动运行时，指定多个Master地址，它们之间用逗号分开，如下所示：

```
MASTER=spark://192.168.100.101:7077,spark://192.168.100.102:7077 bin/spark-shell
```

在ZooKeeper模式下，恢复期间新任务无法提交，已经运行的任务不受影响。

此外，Spark Master还支持一种更简单的单点模式下的错误恢复，即当Master进程异常时，重启Master进程并从错误中恢复。具体方法是设置`spark.deploy.recoveryMode`属性的值为`FILESYSTEM`，并为`spark.deploy.recoveryDirectory`属性设置一个本地目录，用于存储必要的信息来进行错误恢复。

3.3.3 Slave 节点失效

Slave节点运行着Worker、执行器和Driver程序，所以我们分三种情况讨论下3个角色分别退出的容错过程。

- Worker异常停止时，会先将自己启动的执行器停止，Driver需要有相应的程序来重启Worker进程。
- 执行器异常退出时，Driver没有在规定时间内收到执行器的`StatusUpdate`，于是Driver会将注册的执行器移除，Worker收到`LaunchExecutor`指令，再次启动执行器。
- Driver异常退出时，一般要使用检查点重启Driver，重新构造上下文并重启接收器。第一步，恢复检查点记录的元数据块。第二步，未完成作业的重新形成。由于失败而没有处理完成的RDD，将使用恢复的元数据重新生成RDD，然后运行后续的Job重新计算后恢复。

3.4 监控管理

有多种方式来监控Spark程序的运行状况：Web界面、Metrics，还有外部系统。

3.4.1 Web 界面

每个Driver的`SparkContext`都会启动一个Web界面，默认端口是4040，用于显示程序的许多有用信息，包括：

- 调度器Stage、Task列表；
- RDD大小和内存使用统计概况；
- 环境信息；
- 正在运行的执行器信息。

在浏览器中输入地址`http://<driver-node>:4040`就可以访问这个界面，如果同一个节点上运行多个`SparkContext`，它们会绑定在紧接着4040之后的端口上（4041和4042等）。

本地启动./bin/spark-shell后，在浏览器中可以看到如图3-3所示的内容。

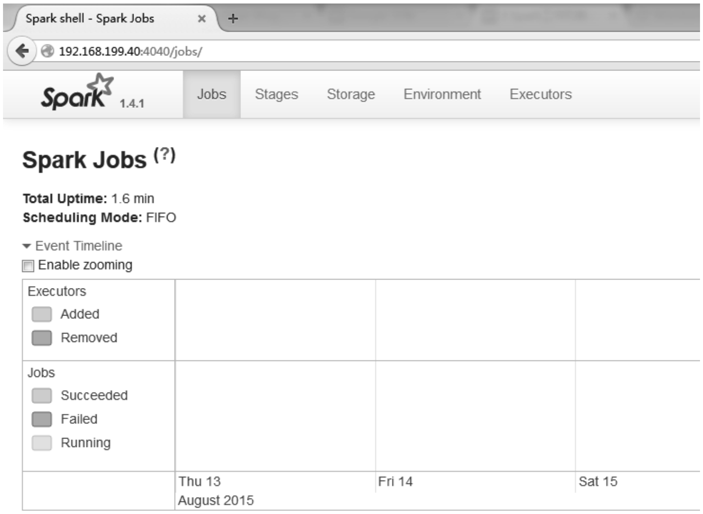
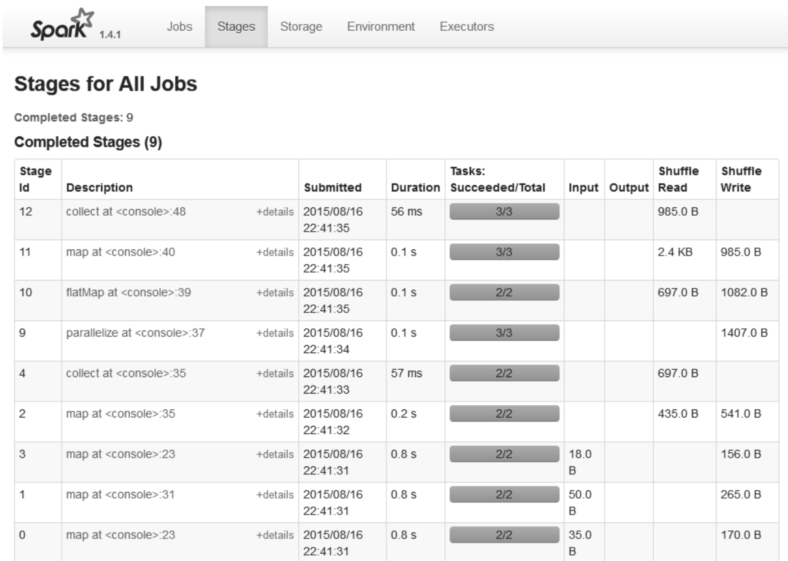


图3-3　本地启动./bin/spark-shell

执行了一些任务后，你将会看到如图3-4所示的内容。



Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
12	collect at <console>:48 +details	2015/08/16 22:41:35	56 ms	3/3			985.0 B	
11	map at <console>:40 +details	2015/08/16 22:41:35	0.1 s	3/3			2.4 KB	985.0 B
10	flatMap at <console>:39 +details	2015/08/16 22:41:35	0.1 s	2/2			697.0 B	1082.0 B
9	parallelize at <console>:37 +details	2015/08/16 22:41:34	0.1 s	3/3				1407.0 B
4	collect at <console>:35 +details	2015/08/16 22:41:33	57 ms	2/2			697.0 B	
2	map at <console>:35 +details	2015/08/16 22:41:32	0.2 s	2/2			435.0 B	541.0 B
3	map at <console>:23 +details	2015/08/16 22:41:31	0.8 s	2/2	18.0 B			156.0 B
1	map at <console>:31 +details	2015/08/16 22:41:31	0.8 s	2/2	50.0 B			265.0 B
0	map at <console>:23 +details	2015/08/16 22:41:31	0.8 s	2/2	35.0 B			170.0 B

图3-4　执行一些任务后的界面

默认情况下，这些信息只能在程序运行时看到。如果希望在程序运行结束之后还能看到这些信息，你需要在程序启动前设置`spark.eventLog.enabled`为`true`，这样这些运行信息会被作

为事件永久存储。

Spark Standalone模式的集群管理器有自己的Web界面。如果Spark程序设置了`spark.eventLog.enabled`为`true`，那么在Spark程序运行结束之后，在集群的Web界面上，还可以看到根据记录的消息重现的Spark程序界面。

如果Spark程序运行在Mesos或YARN上，你还是可以通过Spark历史服务在事后重建程序界面。启动Spark历史服务器的命令如下：

```
./sbin/start-history-server.sh
```

如果`spark.history.provider`是文件系统，则日志根目录必须在`spark.history.fs.logDirectory`中配置，每个子目录对应一个Spark程序的事件日志。历史服务会启动一个Web界面，默认地址是`http://<server-url>:18080`。

3

历史服务支持的配置如表3-3和表3-4所示。

表3-3 历史服务支持的环境变量

环境变量	默认值	作　用
SPARK_DAEMON_MEMORY	512m	分配给历史服务器的内存大小
SPARK_DAEMON_JAVA_OPTS	none	JVM选项
SPARK_PUBLIC_DNS	none	历史服务器的外部域名，可以在公网上访问。如果没有配置，只能通过内部地址引用server，可以导致断链
SPARK_HISTORY_OPTS	none	spark.history.*选项

表3-4 历史服务支持的属性

属　性	默认值	作　用
spark.history.provider	org.apache.spark.deploy.history.FsHistoryProvider	后端实现的类名，目前只有Spark提供的一个实现，从文件系统中查找程序日志
spark.history.fs.logDirectory	file:/tmp/spark-events	存储程序事件日志的根目录，历史服务器从这里加载
spark.history.fs.update.interval	10s	历史服务器更新显示信息的频率，每次都会检查日志的任何变动
spark.history.retainedApplications	50	最多保留的程序界面数量，如果超过，则最旧的程序会被移除
spark.history.ui.port	18080	Web界面的绑定端口
spark.history.kerberos.enabled	false	历史服务器是否使用kerberos来登录。如果历史服务器访问有案例配置的HDFS集群，就需要它了。如果该属性设置为true，就会使用选项spark.history.kerberos.principal和spark.history.kerberos.keytab
spark.history.kerberos.principal	(none)	历史服务器使用的Kerberos主体名称
spark.history.kerberos.keytab	(none)	kerberos keytab文件的位置

（续）

属　性	默认值	作　用
spark.history.ui.acls.enable	false	配置是否对访问用户进行acl授权检查。如果开启，即便这个程序的spark.ui.acls.enable选项没有开启，还是会进行访问控制。Spark程序的所有者永远都有权限访问，spark.ui.view.acls中指定的用户在程序运行时也有权限访问 如果不开启，则不做任何访问控制，所有人都可以访问
spark.history.fs.cleaner.enabled	false	历史服务器是否定期清理事件日志
spark.history.fs.cleaner.interval	1d	旧的事件日志的清理周期，只有比spark.history.fs.cleaner.maxAge还久远的事件日志才被删除
spark.history.fs.cleaner.maxAge	7d	比这个时间还久远的旧日志会被删除

另外，这些Web界面表格的头部都支持点击排序，方便我们定位最慢的Task、数据异常等。

还有，历史服务器只显示已经结束的Spark Job。一个触发Spark程序结束的方法是停止SparkContext对象的运行（方法是sc.stop()），或者在Python中使用with SparkContext() as sc:来处理SparkContext的创建和销毁，这样可以在历史服务器界面上看到这些Job。

3.4.2　REST API

除了通过Web界面查看外，还可以通过JSON接口，这样开发人员可以自己开发可视化展示形式。

历史服务器和正在运行的Spark程序的Web界面都支持REST API。入口都是/api/v1，比如历史服务器的访问地址是http://<server-url>:18080/api/v1，正在运行的Spark程序的访问地址是http://localhost:4040/api/v1。入口地址及对应的意义参见表3-5。

表3-5　入口地址及对应意义

入口地址	意　义
/applications	所有程序的列表
/applications/[app-id]/jobs	指定程序的Job列表
/applications/[app-id]/jobs/[job-id]	指定Job详情
/applications/[app-id]/stages	指定程序的Stage列表
/applications/[app-id]/stages/[stage-id]	指定Stage的attempt列表
/applications/[app-id]/stages/[stage-id]/[stage-attempt-id]	指定Stage attempt的详情
/applications/[app-id]/stages/[stage-id]/[stage-attempt-id]/taskSummary	指定Stage attempt的所有Task的统计信息
/applications/[app-id]/stages/[stage-id]/[stage-attempt-id]/taskList	指定Stage attempt的Task列表
/applications/[app-id]/executors	指定程序的执行器列表
/applications/[app-id]/storage/rdd	指定程序的RDD列表
/applications/[app-id]/storage/rdd/[rdd-id]	指定RDD的存储详情

当运行在YARN上时，每个程序都有多个attempt，所以所有[app-id]实际上是[app-id]/[attempt-id]。

所有这些入口地址都有严格的版本控制，这让基于它进行二次开发更简单。Spark保证：

- 这些地址永远不会在某个版本中删除，任何地址中提供的每个字段都不会被删除；
- 可能添加新的入口地址；
- 现有入口地址可能添加新的字段；
- 每个入口地址在未来都可能添加新版本的API（比如api/V2），新版本不要求向下兼容老版本；
- API版本可能被废弃，但前提是新的版本可以与之共存，而且新版本不只是发布主版本，至少有一个次版本发布。

需要提醒的是，即使是集群上只有一个程序在跑，入口地址中的applications/[app-id]部分也是必不可少的，比如为了查看这个正在运行的程序的所有Job，必须通过http://localhost:4040/api/v1/applications/[app-id]/jobs这个地址。这样可以确保在任何模式下的路径都一致。

3.4.3 Metrics 指标体系

除了Web界面和REST API服务外，Spark还支持基于Coda Hale Metrics Library的指标体系，可以主动将运行状态发送给其他系统，方便与其他监控系统进行集成，比如Ganglia。

Spark支持的Metrics实例有Master、Applications、Worker、执行器和Driver，支持的接收者类型包含在`org.apache.spark.metrics.sink`包中，包括ConsoleSink、CSVSink、JmxSink、MetricsServlet、GraphiteSink和Slf4jSink。

详情请参考官方文档以及Coda Hale Metrics Library文档，这里不作详述。

3.4.4 其他监控工具

除了系统自带的工具，还有其他工具可以帮助我们监控Spark集群的工作状况，具体如下。

- **集群级别**。比如，可以使用Ganglia来监控各节点的CPU、磁盘、网络等负载情况。
- **操作系统级别**。可以使用操作系统自带的工具（比如dstat、iostat、iotop等）来针对单个节点的问题进行定位。
- **Java虚拟机工具**。比如jstack、jmap、jstat、jconsole等。

3.5 Spark 程序配置管理

Spark对程序提供了非常灵活的配置方式，可以使用环境变量、配置文件、命令行参数，还可以直接在Spark程序中指定，不同的配置方式有不同的优化级，可以相互覆盖。而且这些配置

的属性在Web界面中可以直接看到，非常方便我们管理配置。

3.5.1　Spark 程序配置加载过程

Spark程序一般都是由脚本bin/spark-submit来提交的，交互式编程bin/spark-shell其实也是通过它来提交的。通过这种方式启动的Spark程序的加载配置过程如下。

(1) 设置`SPARK_HOME`的值为bin/spark-submit脚本所在目录的上一级目录。

(2) 计算配置文件目录，从环境变量`SPARK_CONF_DIR`中读取。如果没有设置，则取默认值`${SPARK_HOME}/conf`。

(3) 执行配置文件目录下的shell脚本配置文件spark-env.sh，设置基本的环境变量。

(4) 加载配置文件目录下的默认配置文件spark-defaults.conf。

(5) 读取命令行参数，覆盖前面的默认配置。

(6) 使用`SparkConf`对象中的选项，覆盖前面的配置。

上面的加载过程中涉及的Spark程序配置项有两类，一类是第(1)步至第(3)步中使用的环境变量，另外一类是第(4)步至第(6)步中加载的Spark属性项。

3.5.2　环境变量配置

少量基础的Spark程序配置可以通过环境变量的方式来指定，比如配置文件目录是通过环境变量`SPARK_CONF_DIR`来指定的，其默认值是`${SPARK_HOME}/conf`。我们可以在提交Spark程序之前通过指定`SPARK_CONF_DIR`值的方式来使用其他目录作为配置文件的目录。

环境变量可以在提交程序之前通过`export`的方式设置，也可以在配置文件目录下的spark-env.sh文件中指定，其中spark-env.sh本身也是一个脚本。

常用的通用配置项如下所示。

- **`SPARK_LOCAL_IP`**。绑定的IP地址。
- **`SPARK_PUBLIC_DNS`**。Driver程序使用的DNS服务器。
- **`SPARK_CLASSPATH`**。额外追加的`classpath`。

更多环境变量可以参考安装路径下的文件conf/spark-env.sh.template，具体与部署有关。

3.5.3　Spark 属性项配置

如3.5.1节所述，Spark属性项的配置可以在3个地方进行，优先级从低至高依次为：

- spark-defaults.conf
- 命令行参数
- `SparkConf`对象

spark-defaults.conf的初始值可以从`spark-defaults.conf.template`复制而来,示例如下:

```
# Example:
spark.master                     spark://<master IP>:7077
spark.driver.memory              5g
spark.executor.extraJavaOptions  -XX:+PrintGCDetails -Dkey=value -Dnumbers="one two
three"
```

配置中时间相关值的格式如下（不区分大小写）。

- 毫秒，比如25ms。
- 秒，比如5s。
- 分钟，比如10m或10min。
- 小时，比如3h。
- 天，比如5d。
- 年，比如1y。

配置中大小相关值的格式如下（同样不区分大小写）:

- 1B（字节）
- 1K 或 1KB（1 KB = 1024 B）
- 1M 或 1MB（1 MB = 1024 KB）
- 1G 或 1GB（1 GB = 1024 MB）
- 1T 或 1TB（1 TB = 1024 GB）
- 1P 或 1PB（1 PB = 1024 TB）

提交程序时，也可以通过命令行的方式指定参数，且优先级大于配置文件，示例如下：

```
./bin/spark-submit --name "My app" --master local[4] --conf spark.shuffle.spill=false

    --conf "spark.executor.extraJavaOptions=-XX:+PrintGCDetails
-XX:+PrintGCTimeStamps" myApp.jar
```

在Spark程序中也可以指定配置项，而且优先级最高，不过灵活性不如前面的两种方式，每次改动配置需要更新代码并编译。Spark程序中通过`SparkConf`对象来设置配置项，示例如下：

```
val conf = new SparkConf()
    .setMaster("local[2]")
    .setAppName("CountingSheep")
    .set("spark.executor.memory", "1g")
val sc = new SparkContext(conf)
```

常用的配置项参见表3-6。

表3-6　常用的配置项

属性名	默认值	说　　明
`spark.app.name`	(none)	Spark程序名，会出现在Web界面和日志中，一般通过命令行参数指定或直接在程序中指定
`spark.driver.cores`	1	cluster模式下Driver程序使用的core的数量
`spark.driver.maxResultSize`	1 GB	限制Action操作结果返回给Driver程序的最大大小，至少1 MB，如果为0表示不限制。如果超过了这个值，Job会执行失败。设置过高可能导致Driver程序出现out-of-memory错误，注意参考`spark.driver.memory`的取值以及JVM
`spark.driver.memory`	512 MB	Driver程序可以使用的最大内存数量
`spark.executor.memory`	512 MB	每个执行器进程可以使用的最大内存数量
`spark.master`	(none)	集群管理器地址

完整的属性项列表可以参考官方文档。

3.5.4　查看当前的配置

Spark程序的Web界面（详见`http://<driver>:4040`）上的Environment标签显示了所有配置的Spark属性，包括通过spark-defaults.conf、命令行参数以及`SparkConf`对象设置的属性，如图3-5所示。其他没有配置的属性，可以参考官方文档中该属性的默认值。

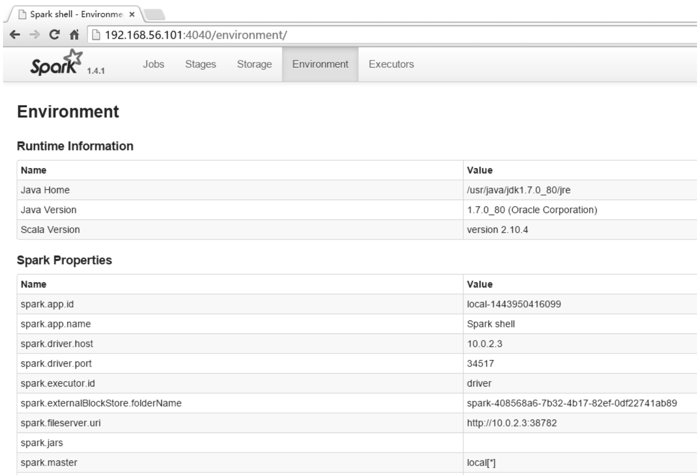


图3-5　Environment标签

3.5.5　配置 Spark 日志

日志配置是另外一项单独的配置，它使用配置文件目录下的log4j.properties作为配置文件，详情可以参考默认安装路径下的conf/log4j.properties.template，你可以复制一份作为默认配置。

第 4 章

Spark内核讲解

本章深入介绍Spark的内核实现，学习内核的过程中难免涉及源码阅读。本书所有的演示都基于Spark 1.4.1，故源码也基于版本1.4.1。读者可以下载下来阅读，也可以通过浏览器直接在GitHub上阅读，地址是https://github.com/apache/spark。

4.1 Spark 核心数据结构 RDD

RDD是Spark最重要的抽象，掌握了RDD，可以说就掌握了Spark计算的精髓。它不但对理解现有Spark程序大有帮助，也能提升Spark程序的编写能力。

RDD的全称是“弹性分布式数据集”（Resilient Distributed Dataset）。首先，它是一个数据集，就像Scala语言中的`Array`、`List`、`Tuple`、`Set`、`Map`也是数据集合一样，但从操作上看RDD最像`Array`和`List`，里面的数据都是平铺的，可以顺序遍历。而且`Array`、`List`对象拥有的许多操作RDD对象也有，比如`flatMap`、`map`、`filter`、`reduce`、`groupBy`等。

其次，RDD是分布存储的。里面的成员被水平切割成小的数据块，分散在集群的多个节点上，便于对RDD里面的数据进行并行计算。

最后，RDD的分布是弹性的，不是固定不变的。RDD的一些操作可以被拆分成对各数据块直接计算，不涉及其他节点，比如`map`。这样的操作一般在数据块所在的节点上直接进行，不影响RDD的分布，除非某个节点故障需要转换到其他节点上。但是在有些操作中，只访问部分数据块是无法完成的，必须访问RDD的所有数据块。比如`groupBy`，在做`groupBy`之前完全不知道每个`key`的分布，必须遍历RDD的所有数据块，将具有相同`key`的元素汇聚在一起，这样RDD的分布就完全重组，而且数量也可能发生变化。此外，RDD的弹性还表现在高可靠性上。

此外，RDD还具有如下特点。

- RDD是只读的，一旦生成，内容就不能修改了。这样的好处是让整个系统的设计相对简单，比如并行计算时不用考虑数据互斥的问题。
- RDD可指定缓存在内存中。一般计算都是流水式生成、使用RDD，新的RDD生成之后，旧的不再使用，并被Java虚拟机回收掉。但如果后续有多个计算依赖某个RDD，我们可以

让这个RDD缓存在内存中，避免重复计算。这个特性在机器学习等需要反复迭代的计算场景下对性能的提升尤其明显。

- RDD可以通过重新计算得到。RDD的高可靠性不是通过复制来实现的，而是通过记录足够的计算过程，在需要时（比如因为节点故障导致内容失效）重新从头或从某个镜像重新计算来恢复的。

4.1.1　RDD 的定义

一个RDD对象，包含如下5个核心属性。

- 一个分区列表，每个分区里是RDD的部分数据（或称数据块）。
- 一个依赖列表，存储依赖的其他RDD。
- 一个名为compute的计算函数，用于计算RDD各分区的值。
- 分区器（可选），用于键/值类型的RDD，比如某个RDD是按散列来分区。
- 计算各分区时优先的位置列表（可选），比如从HDFS上的文件生成RDD时，RDD分区的位置优先选择数据所在的节点，这样可以避免数据移动带来的开销。

下面我们直接来看看这5个属性的具体代码定义。

分区与依赖：

```
// 依赖关系定义在一个Seq数据集中，类型是Dependency
// 有检查点时，这些信息会被重写，指向检查点
private var dependencies_ : Seq[Dependency[_]] = null
// 分区定义在Array数据中，类型是Partition，没用Seq，这主要考虑到随时需要通过下标来访问或更新
// 分区内容，而dependencies_使用Seq是因为它的使用场景一般是取第一个成员或遍历
@transient private var partitions_ : Array[Partition] = null
```

计算函数：

```
/**
 * compute方法由子类来实现，对输入的RDD分区进行计算
 */
def compute(split: Partition, context: TaskContext): Iterator[T]
```

分区器：

```
/** 可选，子类可以重写以指定新的分区方式。Spark支持两种分区方式：Hash和Range*/
@transient val partitioner: Option[Partitioner] = None
```

优先计算位置：

```
/**
 * 可选，子类可以指定分区优先的位置，比如HadoopRDD会重写此方法，让分区尽可能与数据在相同的节点上
 */
protected def getPreferredLocations(split: Partition): Seq[String] = Nil
/**
 * RDD提供统一的调用方法，统一处理检查点问题
```

```
 */
final def preferredLocations(split: Partition): Seq[String] = {
    checkpointRDD.map(_.getPreferredLocations(split)).getOrElse {
        getPreferredLocations(split)
    }
}
```

而且，Spark从第一个开源版本0.3-scala-2.8开始，到目前最新的1.4.1，RDD一直使用这5个核心属性，没有增加，也没减少。可以说，这就是Spark计算的基因。

Spark调度和计算都基于这5个属性，各种RDD都有自己实现的计算，用户也可以方便地实现自己的RDD，比如从一个新的存储系统中读取数据。

图4-1显示了RDD及其常见子类的继承关系。

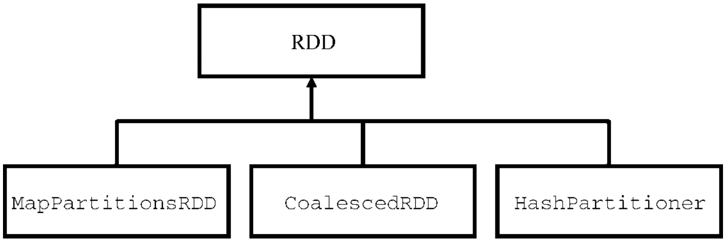


图4-1 RDD及其常见子类的继承关系

RDD完整的定义位于源文件：https://github.com/apache/spark/blob/v1.4.1/core/src/main/scala/org/apache/spark/rdd/RDD.scala。

每个Transformation操作都会生成一个新的RDD，不同操作也可能返回相同类型的RDD，只是计算方法等参数不同。比如，`map`、`flapMap`、`filter`这3个操作都会生成`MapPartitionsRDD`类型的RDD：

```
/**
 * Transformation: map
 */
def map[U: ClassTag](f: T => U): RDD[U] = withScope {
    val cleanF = sc.clean(f)
    new MapPartitionsRDD[U, T](this, (context, pid, iter) => iter.map(cleanF))
}
/**
 *  Transformation: flatMap
 */
def flatMap[U: ClassTag](f: T => TraversableOnce[U]): RDD[U] = withScope {
    val cleanF = sc.clean(f)
    new MapPartitionsRDD[U, T](this, (context, pid, iter) => iter.flatMap(cleanF))
}
/**
 * Transformation: filter
 */
```

```
def filter(f: T => Boolean): RDD[T] = withScope {
    val cleanF = sc.clean(f)
    new MapPartitionsRDD[T, T](
        this,
        (context, pid, iter) => iter.filter(cleanF),
        preservesPartitioning = true)
}
```

4.1.2　RDD 的 Transformation

RDD的Transformation是指由一个RDD生成新RDD的过程，比如前面使用的`flapMap`、`map`、`filter`操作都返回一个新的RDD对象，类型是`MapPartitionsRDD`，它是RDD的子类。

所有的RDD Transformation都只是生成了RDD之间的计算关系以及计算方法，并没有进行真正的计算。下面还是以WordCount为例进行介绍：

```
val textFile = sc.textFile("README.md")
val words = textFile.flatMap(line => line.split(" "))
val wordPairs = words.map(word => (word, 1))
val wordCounts = wordPairs.reduceByKey((a, b) => a + b)
wordCounts.collect()
```

每一个操作都会生成一个新的RDD对象（其类型为RDD子类），它们按照依赖关系串在一起，像一个链表（其实是DAG的简化形式），每个对象有一个指向父节点的指针，以及如何从父节点通过计算生成新对象的信息。图4-2显示了WordCount计算过程中的RDD Transformation生成的RDD对象的依赖关系。

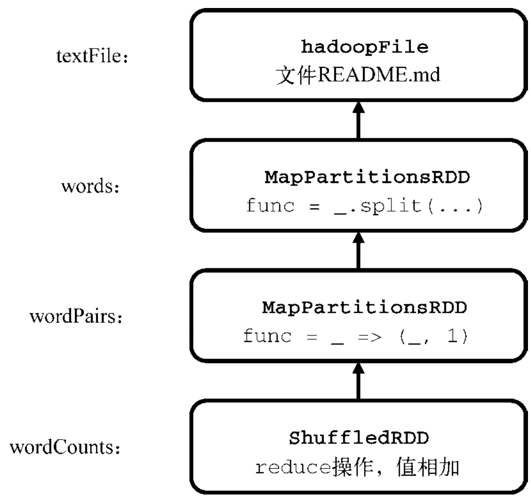


图4-2　RDD Transformation生成的RDD对象的依赖关系

除了RDD创建过程会生成新的RDD外，RDD Transformation也会生成新的RDD，并且设置与

前一个RDD的依赖关系。结合每一个RDD的数据和它们之间的依赖关系，每个RDD都可以按依赖链追溯它的祖先，这些依赖链接就是RDD重建的基础。因此，理解了RDD依赖，也就理解了RDD的重建容错机制。

下面以map为例进行介绍。实际上，这就是生成了一个新的RDD对象，其类型是MapPartitionsRDD（它是RDD的子类）：

```
def map[U: ClassTag](f: T => U): RDD[U] = withScope {
    val cleanF = sc.clean(f)
    new MapPartitionsRDD[U, T](this, (context, pid, iter) => iter.map(cleanF))
}
```

MapPartitionsRDD的定义如下：

```
private[spark] class MapPartitionsRDD[U: ClassTag, T: ClassTag](
    prev: RDD[T],
    f: (TaskContext, Int, Iterator[T]) => Iterator[U],  // (TaskContext, partition
index, iterator)
    preservesPartitioning: Boolean = false)
    extends RDD[U](prev) {
    override val partitioner = if (preservesPartitioning)
        firstParent[T].partitioner else None
    override def getPartitions: Array[Partition] = firstParent[T].partitions
    override def compute(split: Partition, context: TaskContext): Iterator[U] =
        f(context, split.index, firstParent[T].iterator(split, context))
}
```

可以看到，MapPartitionsRDD最主要的工作是用变量f保存传入的计算函数，以便compute调用它来进行计算。其他4个重要属性基本保持不变：分区和优先计算位置没有重新定义，保持不变，依赖关系默认依赖调用的RDD，分区器优先使用上一级RDD的分区器，否则为None。

在Spark中，RDD是有依赖关系的，这种依赖关系有两种类型。

- **窄依赖**。依赖上级RDD的部分分区。
- **Shuffle依赖**。依赖上级RDD的所有分区。

对应类的关系如图4-3所示。

4

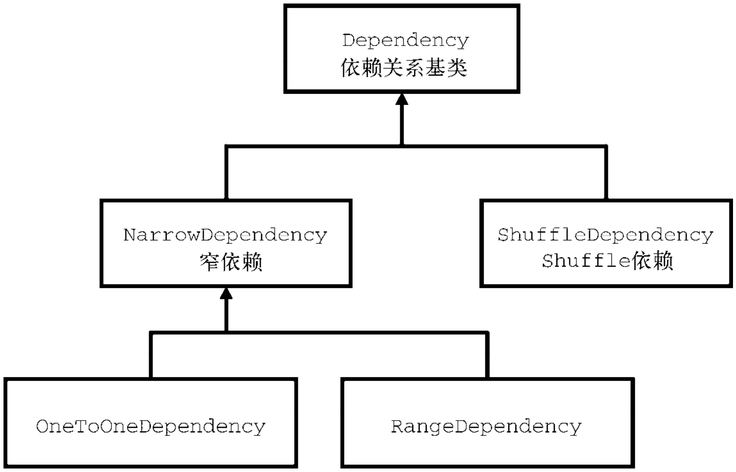


图4-3 对应类的关系

之所以这么区分依赖关系，是因为它们之间有本质的区别。使用窄依赖时，可以精确知道依赖的上级RDD的分区。一般情况下，会选择与自己在同一节点的上级RDD分区，这样计算过程都在同一节点进行，没有网络IO开销，非常高效，常见的`map`、`flatMap`、`filter`操作都是这一类。而Shuffle依赖则不同，无法精确定位依赖的上级RDD的分区，相当于依赖所有分区（想想`reduceByKey`计算，需要对所有的`key`重新排列）。计算时涉及所有节点之间的数据传输，开销巨大。所以，以Shuffle依赖为分隔，Task被分成Stage，方便计算时的管理。

RDD仔细维护着这种依赖关系和计算方法，使得通过重新计算来恢复RDD成为可能。当然，这也不是万能的。如果依赖链条太长，那么通过计算来恢复的代价就太大了。所以，Spark又提供了一种叫检查点的机制。对于依赖链条太长的计算，对中间结果存一份快照，这样就不需要从头开始计算了。比如，第6章将要提到的Spark Streaming流式计算，程序需要7×24小时运行，依赖链接会无穷扩充，如果没有检查点机制，容错将完全没有意义。

4.1.3 RDD 的 Action

RDD的Action是相对Transformation的另一种操作。Transformation代表计算的中间过程，从一个RDD生成新的RDD；而Action代表计算的结束，一次Action调用之后，不再生成新的RDD，结果返回到Driver程序。

鉴于Action具有这样的特点，所以Action操作是不可以在RDD Transformation内部调用的。比如，下面的调用是不允许的：

```
rdd1.map(x => rdd2.values.count() * x)
```

Transformation只是建立计算关系，而Action才是实际的执行者。每个Action都会调用`SparkContext`的`runJob`方法向集群正式提交请求，所以每个Action对应一个Job。比如在`count`

的实现中，先提交Job去集群上运行，返回结果到Driver程序，然后调用sum方法获取数量：

```
/**
 * 返回RDD中的元素数RDD
 */
def count(): Long = sc.runJob(this, Utils.getIteratorSize _).sum
```

4.1.4 Shuffle

Shuffle的概念来自Hadoop的MapReduce计算过程。当对一个RDD的某个分区进行操作而无法精确知道依赖前一个RDD的哪个分区时，依赖关系变成了依赖前一个RDD的所有分区。比如，几乎所有<key, value>类型的RDD操作，都涉及按key对RDD成员进行重组，将具有相同key但分布在不同节点上的成员聚合到一个节点上，以便对它们的value进行操作。这个重组的过程就是Shuffle操作。因为Shuffle操作会涉及数据的传输，所以成本特别高，而且过程复杂。

下面以reduceByKey为例来介绍。在进行reduce操作之前，单词“Spark”可能分布在不同的机器节点上，此时需要先把它们汇聚到一个节点上，这个汇聚的过程就是Shuffle，如图4-4所示。

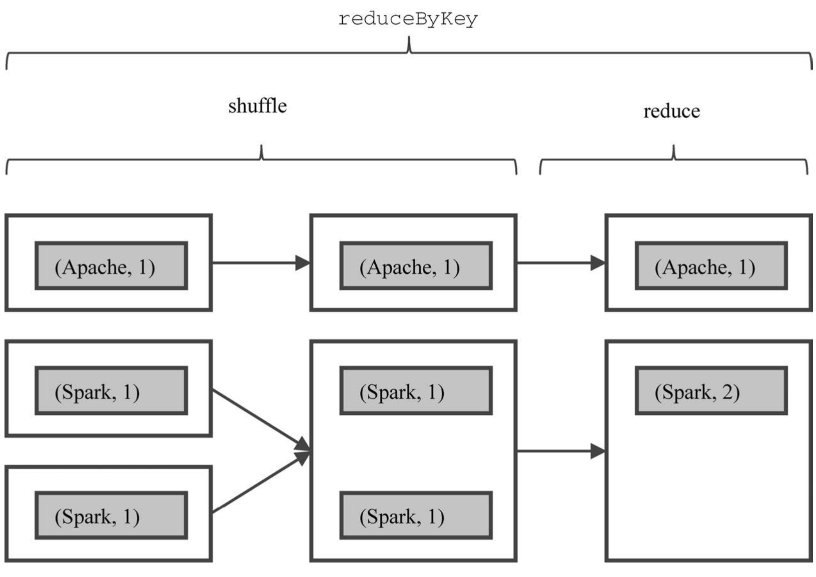


图4-4 Shuffle操作

Shuffle是一个非常消耗资源的操作，除了会涉及大量网络IO操作并使用大量内存外，还会在磁盘上生成大量临时文件，以避免错误恢复时重新计算。因为Shuffle操作的结果其实是一次调度的Stage的结果，而一次Stage包含许多Task，缓存下来还是很划算的。Shuffle使用的本地磁盘目录由spark.local.dir属性项指定。

4.2 SparkContext

SparkContext是Spark程序最主要的入口，用于与Spark集群连接。与Spark集群的所有操作，都通过SparkContext来进行。使用它，可以在Spark集群上创建RDD、计数器以及广播变量。

所有的Spark程序都必须创建一个SparkContext对象。进行流式计算时使用的StreamingContext，以及进行SQL计算时使用的SQLContext，也会关联一个现有的SparkContext或者隐式创建一个SparkContext。

SparkContext类的定义位于core/src/main/scala/org/apache/spark/SparkContext.scala。

4.2.1 SparkConf 配置

初始化SparkContext时，只需要一个SparkConf配置对象作为参数即可。SparkContext类的定义如下：

```
class SparkContext(config: SparkConf)
```

而且，在类定义中，还提供了免参数的构造方法，它会自动生成一个默认的配置：

```
/**
 * 使用默认属性创建一个SparkContext对象
 */
def this() = this(new SparkConf())
```

当然，SparkContext也提供了更多的构造函数来方便我们一次性初始化更多常用选项，比如可以指定master和appName：

```
/**
 * 另外的构造函数，直接设置一个常用属性
 * 参数说明：
 *  master：集群地址，比如spark://host:port、local[4]
 *  appName：程序名，会显示在集群Web UI上
 *  sparkHome：Spark在机器上的安装目录
 *  jars：给集群添加额外的JAR文件集合，可以本地文件，或者是HDFS、HTTP、HTTPS、FTP等类型的URL

 *  environment：环境变量
 */
def this(
    master: String,
    appName: String,
    sparkHome: String = null,
    jars: Seq[String] = Nil,
    environment: Map[String, String] = Map(),
    preferredNodeLocationData: Map[String, Set[SplitInfo]] = Map()) =
    {
    this(SparkContext.updatedConf(new SparkConf(), master,
        appName, sparkHome, jars, environment))
    this.preferredNodeLocationData = preferredNodeLocationData
}
```

保存配置的`SparkConf`类的定义在相同目录下的SparkConf.scala文件中，它最主要的成员是一个散列表，其中`key`和`value`的类型都是字符串类型：

```
private val settings = new ConcurrentHashMap[String, String]()
```

虽然`SparkConf`提供了一些简单的接口来进行配置，但其实所有的配置都以`<key, value>`对的形式保存在`settings`中，比如设置`master`方法其实就是设置了配置项`spark.master`：

```
/**
 * 设置Master地址，比如"local[4]"指向本地或"spark://master:7077"指向一个Spark Standalone
集群
 */
def setMaster(master: String): SparkConf = {
    set("spark.master", master)
}
```

所以，使用配置文件中的`spark.master`配置项，或者参数列表中的`--master`选项，或者`setMaster()`方法，都可以设置Master地址，只是它们的优先级不同。

每个JVM只允许启动一个`SparkContext`，否则默认会抛出异常，比如在通过spark-shell启动的交互式编程环境下，已经默认创建了一个名为`sc`的`SparkContext`对象，如果学习Spark Streming编程时，按照官方文档中的代码直接创建一个`StreamingContext`对象，则会出错：


```
// 刚启动spark-shell时，直接运行下面的代码会出错!!!
import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds, StreamingContext}
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))
```

一个简单的解决办法是先停止默认的`sc`：

```
sc.stop()
```

当然，也可以通过设置`spark.driver.allowMultipleContexts`为`true`来忽略这个错误，这个配置项对应的处理逻辑如下所示：

```
// 如果为true，当多个SparkContexts活动时，只是打印警告日志，而不是抛出异常中止计算
private val allowMultipleContexts: Boolean =
    config.getBoolean("spark.driver.allowMultipleContexts", false)
```

4.2.2 初始化过程

`SparkContext`在构造的过程中，已经完成了各项服务的启动。因为Scala语法的特点，所有构造函数都会调用默认的构造函数，而默认构造函数的代码直接在类定义中。

除了初始化各类配置、日志之外，最重要的初始化操作之一是启动Task调度器和DAG调度器，相关代码如下：

```
// 创建并启动Task调度器
val (sched, ts) = SparkContext.createTaskScheduler(this, master)
_schedulerBackend = sched
_taskScheduler = ts
_dagScheduler = new DAGScheduler(this)
_heartbeatReceiver.send(TaskSchedulerIsSet)
// 创建DAG调度器，并引用之前创建的Task调度器之后，
// 再启动Task调度器
_taskScheduler.start()
```

DAG调度与Task调度的区别是，DAG是最高层级的调度，为每个Job绘制出一个有向无环图（简称DAG），跟踪各Stage的输出，计算完成Job的最短路径，并将Task提交给Task调度器来执行。而Task调度器只负责接受DAG调度器的请求，负责Task的实际调度执行，所以`DAGScheduler`的初始化必须在Task调度器之后。

DAG与Task这种分离设计的好处是，Spark可以灵活设计自己的DAG调度，同时还能与其他资源调度系统结合，比如YARN、Mesos。

Task调度器本身的创建在`createTaskScheduler`函数中进行。根据Spark程序提交时指定的不同模式，可以启动不同类型的调度器。并且出于容错考虑，`createTaskScheduler`会返回一主一备两个调度器。以YARN cluster模式为例，主、备调度器对应不同类的实例，但是加载了相同的配置。下面摘录了`createTaskScheduler`函数的相关实现：

```
private def createTaskScheduler(
    sc: SparkContext,
    master: String): (SchedulerBackend, TaskScheduler) = {
        // 省略部分代码……
        master match {
        // 省略部分case代码……
        case "yarn-standalone" | "yarn-cluster" =>
            if (master == "yarn-standalone") {
                logWarning("\"yarn-standalone\" is deprecated as of Spark 1.0.
                    Use \"yarn-cluster\" instead.")
            }
         // 主调度器
         val scheduler = try {
             val clazz = Class.forName("org.apache.spark.scheduler.
                 cluster.YarnClusterScheduler")
             val cons = clazz.getConstructor(classOf[SparkContext])
             cons.newInstance(sc).asInstanceOf[TaskSchedulerImpl]
         } catch {
             case e: Exception => {
                throw new SparkException("YARN mode not available ?", e)
             }
         }
         // 备用调度器
         val backend = try {
             val clazz =

                  Class.forName("org.apache.spark.scheduler.cluster.
```

```
                YarnClusterScheduler Backend")
            val cons = clazz.getConstructor(classOf[TaskSchedulerImpl],
                classOf[SparkContext])
            cons.newInstance(scheduler, sc).asInstanceOf [CoarseGrainedScheduler
                Backend]
        } catch {
            case e: Exception => {
                throw new SparkException("YARN mode not available ?", e)
            }
        }
        scheduler.initialize(backend)
        (backend, scheduler)
        // 省略部分case代码……
    }
}
```

4.2.3 其他功能接口

SparkContext除了可以初始化环境，连接Spark集群外，还提供了非常多的功能入口，具体如下。

- **创建RDD**。所有的创建RDD的方法都在SparkContext中定义，比如parallelize和textFilenewAPIHadoopFile。
- **RDD持久化**。RDD的持久化操作方法persistRDD、unpersistRDD也在SparkContext中定义。
- **创建共享变量**。包括计数器和广播变量。
- **stop()**。停止SparkContext。
- **runJob**。提交RDD Action操作，这是所有调度执行的入口。

4.3 DAG 调度

SparkContext在初始化时，创建了DAG调度与Task调度来负责RDD Action操作的调度执行。

4.3.1 DAGScheduler

DAGScheduler负责Spark的最高级别的任务调度，调度的粒度是Stage，它为每个Job的所有Stage计算一个有向无环图，控制它们的并发，并找到一个最佳路径来执行它们。具体的执行过程是将Stage下的Task集提交给TaskScheduler对象，由它来提交到集群上去申请资源并最终完成执行。

DAGScheduler的定义位于scheduler/DAGScheduler.scala中。下面是它的类声明，初始化时除了需要一个SparkContext对象外，最重要的是需要输入一个TaskScheduler对象来负责Task的执行：

```
private[spark]
class DAGScheduler(
    private[scheduler] val sc: SparkContext,
    private[scheduler] val taskScheduler: TaskScheduler,
    listenerBus: LiveListenerBus,
    mapOutputTracker: MapOutputTrackerMaster,
    blockManagerMaster: BlockManagerMaster,
    env: SparkEnv,
    clock: Clock = new SystemClock())
extends Logging {
    ...
}
```

1. `runJob`过程

所有需要执行的RDD Action，都会调用`SparkContext.runJob`来提交任务，而`SparkContext.runJob`调用的是`DAGScheduler.runJob`。下面是`runJob`的定义：

```
def runJob[T, U](
    rdd: RDD[T],
    func: (TaskContext, Iterator[T]) => U,
    partitions: Seq[Int],
    callSite: CallSite,
    allowLocal: Boolean,
    resultHandler: (Int, U) => Unit,
    properties: Properties): Unit = {
    val start = System.nanoTime
    val waiter = submitJob(rdd, func, partitions, callSite, allowLocal,
        resultHandler, properties)
    waiter.awaitResult() match {
        case JobSucceeded =>
            logInfo("Job %d finished: %s, took %f s".format
                (waiter.jobId, callSite.shortForm, (System.nanoTime - start) / 1e9))
        case JobFailed(exception: Exception) =>
            logInfo("Job %d failed: %s, took %f s".format
                (waiter.jobId, callSite.shortForm, (System.nanoTime - start) / 1e9))
        throw exception
    }
}
```

`runJob`调用`submitJob`提交任务，并等待任务结束。

任务提交后的处理过程大概如下。

(1) `submitJob`生成新的Job ID，发送消息`JobSubmitted`。

(2) DAG收到`JobSubmitted`消息，调用`handleJobSubmitted`来处理。

(3) `handleJobSubmitted`创建一个`ResultStage`，并使用`submitStage`来提交这个`ResultStage`。

上面的过程看起来没完，实际上大的过程已经结束了，猫腻在`submitStage`中。Spark的执行过程是“懒惰”（lazy）的，这在这里得到了完整的体现。任务提交时，不是按Job的先后顺序

提交的，而是倒序的。每个Job的最后一个操作是Action操作，DAG把这最后的Action操作当作一个Stage，首先提交，然后逆向逐级递归填补缺少的上级Stage，从而生成一棵实现最后Action操作的最短的（因为都是必须的）有向无环图，然后再从头开始计算。

submitStage方法的实现代码如下所示：

```
// 提交Stage之前，先递归提交所缺失的父Stage
private def submitStage(stage: Stage) {
    val jobId = activeJobForStage(stage)
    if (jobId.isDefined) {
        logDebug("submitStage(" + stage + ")")
        if (!waitingStages(stage) && !runningStages(stage) && !failedStages(stage))
        {
            val missing = getMissingParentStages(stage).sortBy(_.id)
            logDebug("missing: " + missing)
            if (missing.isEmpty) {
                // 仅在所有缺失的父Stage都提交执行了，才开始提交自己
                logInfo("Submitting " + stage + " (" + stage.rdd + "),
                    which has no missing parents")
                submitMissingTasks(stage, jobId.get)
            } else {
                for (parent <- missing) {
                    submitStage(parent)
                }
                waitingStages += stage
            }
        }
    } else {
        abortStage(stage, "No active job for stage " + stage.id)
    }
}
```

可以看到，这是一个逆向递归的过程，先查找所有缺失的上级Stage并提交，待所有上级Stage都提交执行了，才轮到执行当前Stage对应的Task。

查找上级Stage的过程，其实就是递归向上遍历所有RDD依赖列表并生成Stage的过程，代码如下所示：

```
private def getMissingParentStages(stage: Stage): List[Stage] = {
    val missing = new HashSet[Stage]
    val visited = new HashSet[RDD[_]]
    // 这里手工维护一个堆栈，避免递归访问过程中的栈溢出错误
    val waitingForVisit = new Stack[RDD[_]]
    def visit(rdd: RDD[_]) {
        if (!visited(rdd)) {
            visited += rdd
            if (getCacheLocs(rdd).contains(Nil)) {
                for (dep <- rdd.dependencies) {
                    dep match {
                        case shufDep: ShuffleDependency[_, _, _] =>
                            val mapStage = getShuffleMapStage(shufDep, stage.jobId)
                            if (!mapStage.isAvailable) {
```

```
                                missing += mapStage
                            }
                        case narrowDep: NarrowDependency[_] =>
                            waitingForVisit.push(narrowDep.rdd)
                    }
                }
            }
        }
    }
    waitingForVisit.push(stage.rdd)
    while (waitingForVisit.nonEmpty) {
        visit(waitingForVisit.pop())
    }
    missing.toList
}
```

熟悉数据结构的读者不难发现，遍历的过程是非递归的层序遍历（不是前序、中序或后序），使用了一个堆栈来协助遍历，而且保证了层序的顺序与DAG中的依赖顺序一致。

2. Stage

值得注意的是，仅对依赖类型是`ShuffleDependency`的RDD操作创建Stage，其他的RDD操作并没有创建Stage。这里我们补一下Stage的概念。前面讲RDD时提到过，RDD操作有两类依赖：一类是窄依赖，一个RDD分区只依赖上一个RDD的部分分区，而且这些分区都在相同的节点上；另外一类依赖是Shuffle依赖，一个RDD分区可能会依赖上一级RDD的全部分区，一个典型的例子是`groupBy`聚合操作。这两类操作在计算上有明显的区别，窄依赖都在同一个节点上进行计算，而Shuffle依赖跨越多个节点，甚至所有涉及的计算节点。因此，DAG在调度时，对于在相同节点上进行的Task计算，会合并为一个Stage。

所以，只有两种情况下会生成新的Stage：一类是依赖类型是Shuffle的Transformation操作会触发生成新的Stage，几乎所有的`ByKey`操作都是，比如`reduceByKey`和`groupByKey`；另外一类是Action操作，是为了生成默认的Stage，这样即便没有Shuffle类操作，保证至少有一个Stage。

总结一下，各Stage之间以Shuffle为分界线。

4.3.2 TaskScheduler

相对`DAGScheduler`而言，`TaskScheduler`是低级别的调度接口，允许实现不同的Task调度器。目前，已经实现的Task调度器除了自带的以外，还有YARN和Mesos调度器。每个`TaskScheduler`对象只服务于一个`SparkContext`的Task调度。`TaskScheduler`从`DAGScheduler`的每个Stage接收一组Task，并负责将它们发送到集群上，运行它们，如果出错还会重试，最后返回消息给`DAGScheduler`。

`TaskScheduler`的主要接口包括一个钩子接口（也称hook，表示定义好之后，不是用户主动调用的），被调用的时机是在初始化完成之后和调度启动之前：

```
def postStartHook() { }
```

还有启动和停止调度的命令：

```
def start(): Unit
def stop(): Unit
```

此外，还有提交和撤销Task集的命令：

```
def submitTasks(taskSet: TaskSet): Unit
def cancelTasks(stageId: Int, interruptThread: Boolean)
```

第 5 章

Spark SQL与数据仓库

从Spark 1.3开始，Spark SQL正式发布。而之前的另一个基于Spark的SQL开源项目Shark随之停止更新，基于Spark的最佳的SQL计算就是Spark SQL。

Spark SQL是Spark的一个模块，专门用于处理结构化数据。图5-1显示了Spark SQL与Spark核心及其他模块之间的关系。

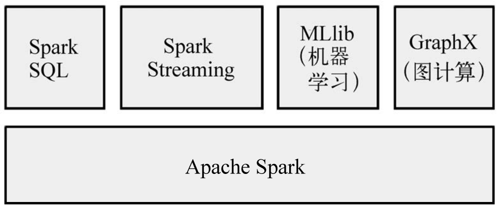


图5-1　Spark SQL与Spark核心及其他模块之间的关系

Spark SQL的最大优势是性能非常高，这得益于Spark的基因，而且还使用了基于成本的优化器、列存储、代码生成等技术。此外，Spark SQL也可以扩展到上千个计算节点以及数小时的计算能力，并且支持自动容错恢复。

Spark提供了传统的SQL交互式查询功能，还支持业界标准的JDBC/ODBC访问接口，这样很容易用Spark SQL来实现分布式数据仓库，以及商业智能计算。而且Spark SQL与Apache Hive基本完全兼容，我们可以像使用Hive一样来使用Spark SQL。

而且Spark SQL还提供领域API，并且提供了专门的数据结构抽象DataFrame，可以让Spark程序轻松进行SQL操作。它支持Scala、Java、Python和R这4种编程语言，比如在Scala代码中做SQL查询：

```
sqlContext = HiveContext(sc)
val df = sqlContext.sql("SELECT * FROM table")
```

此外，Spark SQL支持非常多的数据源，包括Hive、Avro、Parquet、ORC、JSON、JDBC，而且提供了统一的访问形式，使用起来非常方便。

5.1　Spark SQL 基础

使用Spark SQL有两种方式：一种是作为分布式SQL引擎，此时只需要写SQL就可以进行计算，不需要复杂的编码，这也是最受欢迎的一种方式，本节会重点介绍；另外一种模式是在Spark程序中，通过领域API的形式来操作数据（被抽象为DataFrame）。

5.1.1　分布式 SQL 引擎

作为分布式引擎时，有两种运行方式，一种是JDBC/ODBC Server，另一种是使用Spark SQL命令行。在正式环境下，建议使用前者，后者仅在本地测试时使用。

JDBC/ODBC服务的部署结构如图5-2所示。所以，除了前面部署的Spark集群之外，我们还需要部署一个DB用于存储数据库元数据，以及启动一个JDBC/ODBC Server作为对外服务的接口。

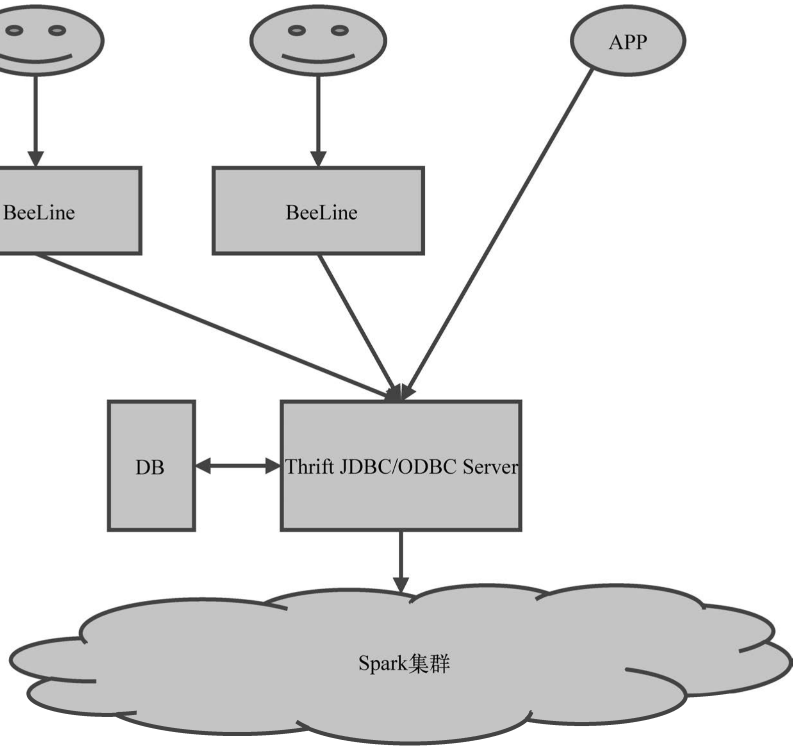


图5-2　JDBC/ODBC服务的部署结构

1. 部署RDB

首先，我们需要部署一个DB，它可以是任意一种关系型数据库，比如MySQL、PostgreSQL。这里以在Ubuntu上部署MySQL为例进行介绍，执行下面的命令即可安装MySQL并启动服务：

```
apt-get install mysql-server
```

而且，还需要给Spark SQL添加账号并开通权限。默认使用的DB名是hiveMetastore，假设账号和密码都是`spark`，授权SQL可以这样写：

```
mysql> grant all on hiveMetastore.* to spark@'localhost' identified by 'spark';
mysql> flush privileges;
```

2. 准备配置文件conf/hive-site.xml

接下来，我们准备启动JDBC/ODBC Server，在启动之前，需要准备一下配置文件。JDBC/ODBC Server是基于Hive 0.13中的HiveServer2来实现的，所以JDBC/ODBC Server的配置方法与HiveServer2的配置方法基本相同。如果是与现有的Hive一起工作，直接使用Hive的配置文件conf/hive-site.xml即可，或者新建一个，在conf目录下准备一个名为hive-site.xml的配置文件，其内容如下：

```
<configuration>
    <property>
        <name>javax.jdo.option.ConnectionDriverName</name>
        <value>com.mysql.jdbc.Driver</value>
        <description>指定JDO驱动类型，这里是mysql</description>
    </property>
    <property>
        <name>javax.jdo.option.ConnectionURL</name>
        <value>
            jdbc:mysql://localhost:3306/hiveMetastore?createDatabaseIfNotExist=true
        </value>
        <description>mysql地址，使用的DB名是hiveMetastore，createDatabaseIfNotExist为
            true表示当hiveMetastore这个DB不存在时，会自动创建hiveMetastore</description>
    </property>
    <property>
        <name>javax.jdo.option.ConnectionUserName</name>
        <value>spark</value>
        <description>mysql账号</description>
    </property>
    <property>
        <name>javax.jdo.option.ConnectionPassword</name>
        <value>spark</value>
        <description>mysql密码</description>
    </property>
</configuration>
```

除了上面的选项之外，还支持更多的选项，具体如下。

- **`hive.server2.thrift.bind.host`**。TCP绑定的地址，默认为localhost，表示所有IP。
- **`hive.server2.thrift.port`**。TCP监听的端口，默认值为10 000。
- **`hive.server2.thrift.min.worker.threads`**。Server线程数的最小值，默认值为5。
- **`hive.server2.thrift.max.worker.threads`**。Server线程数的最大值，默认值为500。

> **提示**　当启动多个JDBC/ODBC Server时，经常出现的问题是，连接到其中一个Server的客户端可以看到某个新创建的表，而连接到其他Server的客户端却看不到。这是由于默认的缓存机器导致的，可以设置选项`datanucleus.cache.level2.type`的值为`none`来关闭缓存。

3. 启动JDBC/ODBC Server

现在可以启动JDBC/ODBC Server了，其命令是：

```
./sbin/start-thriftserver.sh
```

使用`--help`命令可以看到所有的选项列表，它基本上与bin/spark-submit脚本相同，唯一多出的`--hiveconf`可用于设置Hive属性，优先级高于配置文件conf/hive-site.xml。比如，如果我们不想使用默认的或配置文件中指定的绑定IP和端口，就通过`--hiveconf`选项这样设置：

```
./sbin/start-thriftserver.sh \
    --hiveconf hive.server2.thrift.port=<listening-port> \
    --hiveconf hive.server2.thrift.bind.host=<listening-host> \
```

绑定的IP和端口可以通过环境变量的方式来指定，而且优先级高于`--hiveconf`选项，更高于配置文件：

```
export HIVE_SERVER2_THRIFT_PORT=<listening-port>
export HIVE_SERVER2_THRIFT_BIND_HOST=<listening-host>
./sbin/start-thriftserver.sh
```

JDBC/ODBC Server还可以运行在HTTP协议上，这样方便通过代理进行访问，其方法是设置选项`hive.server2.transport.mode`的值为HTTP，默认端口为10001，此时就不支持默认的TCP模式了。关于HTTP模式及JDBC/ODBC Server其他选项的更多内容，可以参考HiveServer2的配置。

4. 使用beeline交互式工具

JDBC/ODBC Server启动之后，我们可以用beeline来测试启动是否正常：

```
./bin/beeline
```

下面的命令可以连接到JDBC/ODBC Server：

```
beeline> !connect jdbc:hive2://localhost:10000
```

此时beeline会要求我们输入用户名和密码，默认是非安全模式，只需要输入用户名和空的密码即可。

也可以在启动beeline时，直接指定JDBC Server：

```
./bin/beeline -u 'jdbc:hive2://localhost:10000'
```

这是beeline的完整选项列表：

```
$ ./bin/beeline --help
Usage: java org.apache.hive.cli.beeline.BeeLine
    -u <database url>                the JDBC URL to connect to
    -n <username>                    the username to connect as
    -p <password>                    the password to connect as
    -d <driver class>                the driver class to use
    -e <query>                       query that should be executed
    -f <file>                        script file that should be executed
    --hiveconf property=value        Use value for given property
    --hivevar name=value             hive variable name and value
                                     This is Hive specific settings in which variables
                                     can be set at session level and referenced in Hive
                                     commands or queries.
    --color=[true/false]             control whether color is used for display
    --showHeader=[true/false]        show column names in query results
    --headerInterval=ROWS;          the interval between which heades are displayed
    --fastConnect=[true/false]      skip building table/column list for tab-completion
    --autoCommit=[true/false]        enable/disable automatic transaction commit
    --verbose=[true/false]           show verbose error messages and debug info
    --showWarnings=[true/false]      display connection warnings
    --showNestedErrs=[true/false]    display nested errors
    --numberFormat=[pattern]         format numbers using DecimalFormat pattern
    --force=[true/false]             continue running script even after errors
    --maxWidth=MAXWIDTH              the maximum width of the terminal
    --maxColumnWidth=MAXCOLWIDTH     the maximum width to use when displaying columns
    --silent=[true/false]            be more silent
    --autosave=[true/false]          automatically save preferences
    --outputformat=[table/vertical/csv/tsv]   format mode for result display
    --isolation=LEVEL                set the transaction isolation level
    --nullemptystring=[true/false]  set to true to get historic behavior of printing
                                     null as empty string
    --help                           display this message
```

如果遇到SQL查询结果中文显示乱码的问题，可以通过修改系统编码的方式解决，比如数据是UTF-8编码，在启动beeline之前，我们可以这样指定终端编码：

```
LANG=zh_CN.UTF-8; ./bin/beeline
```

5. 运行Spark SQL命令行界面

Spark SQL命令行界面（以下简称CLI）是另一个运行Spark SQL的方式，使用它，可以在本地直接启动服务，这非常适合调试使用。注意Spark SQL CLI与Thrift JDBC服务是不同的。

要启动Spark SQL CLI，只需要执行如下命令，不需要像JDBC/ODBC那样先启动一个额外的服务：

```
./bin/spark-sql
```

Spark SQL CLI也支持与JDBC/ODBC Server完全一样的配置。使用`./bin/spark-sql --help`命令，可以查看所有的选项列表。

5.1.2　支持的 SQL 语法

Spark SQL在设计上与Hive兼容，是一种“开箱即用”的兼容方式，即可以直接替代Hive的角色，但内部实现完全不同。可以直接使用Hive的meta数据库，语法上也基本完全兼容HiveQL，不同的地方非常少。一般情况下，完全可以使用HiveQL来操作Spark SQL，发现异常时再通过查询手册等方式来解决。

Spark SQL支持绝大部分的Hive特性，具体如下所示。

Hive查询语句，包括：

- SELECT
- GROUP BY
- ORDER BY
- CLUSTER BY
- SORT BY

所有Hive运算符，包括：

- 关系运算符（=、⇔、==、<>、<、>、>=、<=等）
- 数学运算符（+、-、*、/和%等）
- 逻辑运算符（AND、&&、OR、||等）
- 复杂类型构造器（struct、named_struct）
- 数学函数（sign、ln、cos等）
- 字符串函数（instr、length、printf等）
- 用户自定义函数（UDF）
- 用户自定义聚合函数（UDAF）
- 用户自定义序列化格式（SerDes）
- 求交
    - JOIN
    - {LEFT|RIGHT|FULL} OUTER JOIN
    - LEFT SEMI JOIN
    - CROSS JOIN
- Unions
- 子查询
    - SELECT col FROM (SELECT a + b AS col from t1) t2
- Sampling
- Explain

- 表分区
- View

所有Hive DDL函数，包括：

- CREATE TABLE
- CREATE TABLE AS SELECT
- ALTER TABLE

绝大部分Hive数据类型，包括：

- TINYINT
- SMALLINT
- INT
- BIGINT
- BOOLEAN
- FLOAT
- DOUBLE
- STRING
- BINARY
- TIMESTAMP
- DATE
- ARRAY<>
- MAP<>
- STRUCT<>

从Spark 1.4开始，Spark SQL已经开始支持窗口函数了，包括排名函数rank、dense_rank、percent_rank、ntile、row_number，以及分析函数cume_dist、first_value、last_value、lag、lead。使用时，在函数后面要加Over表达式OVER (PARTITION BY ... ORDER BY ...)。

当然，有一些Hive特性是Spark SQL不支持的，不过这些特性实际上很少用到。最主要的是不支持Hive的bucket表，使用散列的方式对Hive表进行分区。另外，还有一些Hive比较隐秘的特性Spark SQL也不支持，包括UNION类型、Unique join。

值得注意的是，Hive上的优化方法，大部分Spark SQL都不支持。因为Spark SQL只是在使用上与Hive兼容，实现机制完全不一样，一些问题可能在Spark的内存计算模式下也不是问题。

5.1.3　支持的数据类型

Spark SQL和DataFrame支持大部分Hive类型，具体如下所示。

- **数字类型**。ByteType、ShortType、IntegerType、LongType、FloatType、DoubleType

和DecimalType。

- 字符串类型。StringType。
- 二进制类型。BinaryType。
- Bool类型。BooleanType。
- 日期时间类型。TimestampType和DateType。
- 复杂类型。ArrayType、MapType和StructType。

在Spark程序中，这些类型会转换到特定语言的特定类型，而且取值范围会尽量保持不变。如果不能完全对应，会使用值范围更大的类型来表示，但值不变。比如ByteType类型的取值范围是−128~127，在Scala下会使用byte表示，Java中也是byte，取值范围都是−128~127，无缝对接，而Python和R中没有对应的类型，所以用整型表示。

5.1.4 DataFrame

Spark SQL除了支持使用SQL方式来查询之外，还提供了编程接口，可以在Spark程序中引用Spark SQL模块。编程时使用的数据抽象是DataFrame，它具有与RDD类似的分布式数据集特点，但还增加了列的概念，这样可以与传统关系型数据库的表对应起来。与R和Python中的数据框（data frame）在概念上也是相同的，Spark程序支持的Scala、Java、Python以及R这4种编程语言都可以使用DataFrame。

在Spark中使用DataFrame的过程也很简单，大概需要如下4步：

(1) 初始化环境，一般是创建一个SQLContext对象；
(2) 创建一个DataFrame，可以来源于RDD或其他数据源；
(3) 调用DataFrame操作，是一种领域特定的API，可以实现所有的SQL功能；
(4) 或者通过函数直接执行SQL语句。

5

1. 从创建SQLContext开始

就像普通Spark程序需要SparkContext这个对象来保存环境一样，DataFrame的环境对象是SQLContext。Spark SQL相关的所有函数，都在SQLContext或它的子类中。SQLContext的创建方法如下：

```
val sc: SparkContext // sc是已经存在的SparkContext实例
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

除了最基本的SQLContext，我们还可以使用HiveContext，它继承自SQLContext，提供了更多Hive相关的功能，比如读Hive表，访问Hive用户定义函数（UDF）。但在未来，SQLContext会支持所有HiveContext支持的特性，HiveContext将不再必需。

HiveContext这样进行初始化：

```
val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)
```

2. 创建DataFrame

环境初始化之后，就可以创建DataFrame了，主要有两种创建方式，具体如下所示。

- 从RDD创建，又分两种：
  - 用Scala反射，代码简洁，但需要提前知道数据格式；
  - 程序指定，略复杂，但可以运行时指定。
- 从其他数据源创建。

- **使用反射的方法从RDD创建DataFrame**

这个方法是先定义一个`case class`，参数名即为列名，然后将RDD的成员转换成`case class`类型，包含`case class`的RDD可以通过反射方式被隐式转换成DataFrame，`case class`的参数名会成为表的列名，然后就可以注册成一张表。

这种方法使代码更简洁，但前提是你在写程序的时候已经知道了数据格式，可以预行设定表的模式。示例如下：

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
import sqlContext.implicits._
// 定义一个case class，参数名即为表的列名
// 注意，Scala 2.10最多支持22个字段名，如果想绕开这个限制，可以使用自定义class并实现Product接口
case class Person(name: String, age: Int)
// 从文本文件创建RDD
val rdd = sc.textFile("examples/src/main/resources/people.txt").map(_.split(","))
// 还是RDD，但包含了case class
val rddContainingCaseClass = rdd.map(p => Person(p(0), p(1).trim.toInt))
// 包含case class的RDD被隐式转换成DataFrame，注意toDF是DataFrame的方法，而不是RDD的
val people = rddContainingCaseClass.toDF()
// 将DataFrame的内容打印到标准输出
people.show()
```

输出如下：

```
+-------+---+
|   name|age|
+-------+---+
|Michael| 29|
|   Andy| 30|
| Justin| 19|
+-------+---+
```

- **使用程序动态从RDD创建DataFrame**

当`case class`无法提前知道时（例如，记录的结构被编码在字符串中，对字符串进行解析之后才知道结构，或者需要针对不同的用户呈现不同的字段集），我们还可以在运行时动态指定表模式来从RDD创建DataFrame，这个方法需要经过如下这3步。

(1) 从原来的RDD创建一个新的RDD，成员是`Row`类型，包含所有列。

(2) 创建一个StructType类型的表模式，其结构与步骤(1)中创建的RDD的Row结构相匹配。

(3) 使用SQLContext.createDataFrame方法将表模式应用到步骤(1)创建的RDD上。

虽然这种方法略显复杂，但它可以让你在运行时动态创建，示例如下：

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
// 一个普通RDD
val people = sc.textFile("examples/src/main/resources/people.txt")
// 字符串格式的表模式
val schemaString = "name age"
// 导入依赖的数据类型
import org.apache.spark.sql.Row;
import org.apache.spark.sql.types.{StructType, StructField, StringType};
// 根据字符串格式的表模式创建结构化的表模式，用StructType保存
val schema =
    StructType(
        schemaString.split(" ").map(fieldName => StructField(fieldName, StringType,
            true)))
// 将普通RDD的成员转换成Row对象
val rowRDD = people.map(_.split(",")).map(p => Row(p(0), p(1).trim))
// 将模式作用到RDD上，生成DataFrame
val peopleDataFrame = sqlContext.createDataFrame(rowRDD, schema)
// 将DataFrame的内容打印到标准输出
peopleDataFrame.show()
```

- **从其他数据源生成DataFrame**

Spark提供了统一的接口，可以很方便地从其他数据源创建DataFrame，比如从JSON格式的文件创建DataFrame：

```
val df = sqlContext.read.json("examples/src/main/resources/people.json")
df.show()
```

无论是SQL方式和DataFrame编程方式，都支持许多类型的数据源，详情将在下一节中单独讲述。

3. DataFrame操作

DataFrame支持许多特殊的操作，称为领域编程语言或领域API。上面示例中的show方法其实就是一个常用的DataFrame操作，而且这些操作都可以翻译成对应的SQL语句，虽然表现形式不同，但神似。下面是示例：

```
val sc: SparkContext // 已经存在的SparkContext
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
// 创建DataFrame，相当于生成一张表，假设表名是people
val df = sqlContext.read.json("examples/src/main/resources/people.json")
// 将DataFrame的内容打印到标准输出
// 对应的SQL：
// select * from people
df.show()
// age  name
```

```
// null Michael
// 30   Andy
// 19   Justin
// 以树形打印DataFrame模式
// 对应的SQL:
// show create table people
df.printSchema()
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)
// 选择列"name"
// 对应的SQL:
// select name from people
df.select("name").show()
// name
// Michael
// Andy
// Justin
// 选择所有，并且age列加1
// 对应的SQL:
// select name, age +1 from people
df.select(df("name"), df("age") + 1).show()
// name    (age + 1)
// Michael null
// Andy    31
// Justin  20
// 选择age大于21的
// 对应的SQL:
// select * from people where age > 21
df.filter(df("age") > 21).show()
// age name
// 30  Andy
// 按age分组，并输出各组的数量
// 对应的SQL:
// select age, count(*) from people group by age
df.groupBy("age").count().show()
// age  count
// null 1
// 19   1
// 30   1
```

4. 执行SQL

除了上述通过领域API的方式访问DataFrame外，还可以将DataFrame注册成表，然后使用纯SQL语句的方式来访问。

首先，使用registerTempTable方法将DataFrame注册成一张表：

```
df.registerTempTable("people")
```

然后就可以进行SQL查询了：

```
val result = sqlContext.sql("SELECT * FROM people")
```

另外，上面查询使用的SQL支持使用不同的SQL语法解析器。默认情况下，会使用SQLContext中自带的一个非常简单的SQL语法解析器，如果不够用，可以换成功能更强大的HiveQL解析器。这可以使用`spark.sql.dialect`选项重新指定，设置成`hiveql`，可以通过SQLContext中的`setConf`方法设置，或者在SQL中使用`SETkey=value`命令来设置。

当使用HiveContext时，DataFrame还可以使用`saveAsTable`方法持久化，内容会被保存到存储系统中，表信息会保存在数据库中，这样即使程序退出后，其他进程还可以访问它。比如：

```
val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)
val df = sqlContext.read.json("examples/src/main/resources/people.json")
df.saveAsTable("people")
```

如果是在本地模式，数据会保存至/user/hive/warehouse/people/目录下，表信息会保存在metastore_db目录下，通过./bin/spark-sql可以看到default数据下会出现一张新的表people。如果是在集群模式下，上面代码的效果与通过JDBC/ODBC Server创建一张表并导入数据相同，也可以通过beeline看到表people。

5.1.5 DataFrame 数据源

DataFrame支持非常多类型的数据源，包括Hive、Avro、Parquet、ORC、JSON、JDBC。而且Spark提供了统一的读写接口。

通过数据源加载数据时，默认的类型是Parquet，这是一种大数据计算中最常用的列式存储格式。下面以Scala为例进行介绍，其加载方法是：

```
val df = sqlContext.read.load("examples/src/main/resources/users.parquet")
```

对于其他类型，可以使用`format`指定：

```
val df =
    sqlContext.read.format("json").load("examples/src/main/resources/users.json")
```

数据源类型的名字一般应该是全称，比如`org.apache.spark.sql.parquet`，但对于内置类型，我们可以使用缩写，比如Parquet、JSON、JDBC。

保存时，默认类型也是Parquet：

```
df.select("name", "age").write.save("namesAndFavColors.parquet")
```

同样，对于其他类型，使用`format`指定：

```
df.select("name", "age").write.format("parquet").save("namesAndAges.parquet")
```

像普通写文件一样，保存时也有几种模式，具体如下。

- **SaveMode.ErrorIfExists**。默认模式，如果文件已经存在，就报错。
- **SaveMode.Append**。追加模式。

- **SaveMode.Overwrite**。覆盖写模式。
- **SaveMode.Ignore**。如果文件存在就放弃写，这类似SQL中的`CREATE TABLE IF NOT EXISTS`。

5.1.6　性能调优

Spark SQL自身的性能已经非常领先了，个人需要做的并不多。最重要的性能优化方法是，使用内存来缓存数据，调用的方法是`sqlContext.cacheTable("tableName")`或`dataFrame.cache()`。此外，调用`sqlContext.uncacheTable("tableName")`可以从内存中移除数据。

内存缓存可以通过如下两个系统配置来控制。

- **spark.sql.inMemoryColumnarStorage.compressed**。默认值为`true`，表示开启压缩。
- **spark.sql.inMemoryColumnarStorage.batchSize**。缓存块大小，增加它可以提升内存利用率，但也会增加内存溢出的风险。

此外，还有一些选项可以用于优化，但不建议做，未来这些优化工作都将自动进行。

除了系统层面的优化，应用程序本身的优化也是非常重要的环节，需要不断积累经验。

5.2　Spark SQL 原理和运行机制

在Spark出现之前，基于Hadoop MR的Hive一直是开源大数据SQL计算的唯一选择，但其性能一直被人诟病。Spark在刚推出的几年间，并不支持SQL计算，但它相对于Hadoop MR的巨大性能优势还是非常吸引人的，于是出现了基于Spark的SQL计算框架Shark。

Shark基于Hive的代码实现，但底层的计算改为使用Spark，这使得Shark相对于Hive有非常大的性能优势，而且Shark还是第一个支持交互式查询的SQL引擎，非常容易入手。

然而Shark大量使用Hive的代码也带来严重的弊端，这给进一步的优化和维护工作带来非常大的困难，也无法充分利用Spark以及Scala语言的优势，而且对于Spark的集成计算战略也是很大的障碍。

因此，在Shark的基础上，Spark SQL进行了重新开发，语法上尽可能保持与Hive兼容，最大的动作是重写了执行优化器，代号为Catalyst。

相对于Shark，Spark SQL的最大优势是性能。在分析型TPC-DS基准测试中，Spark SQL的性能超过Shark一个数量级，而且还提供了编程接口，可以让普通Spark程序也能享受SQL语言的便利。

由于Spark SQL的优势明显，Shark目前已经停止开发，由Spark SQL正式接任，并在Spark 1.3中正式发布。

5.2.1 Spark SQL 整体架构

Spark SQL同时支持DataFrame编程API，以及SQL执行和JDBC/ODBC接口，整体结构如图5-3所示。

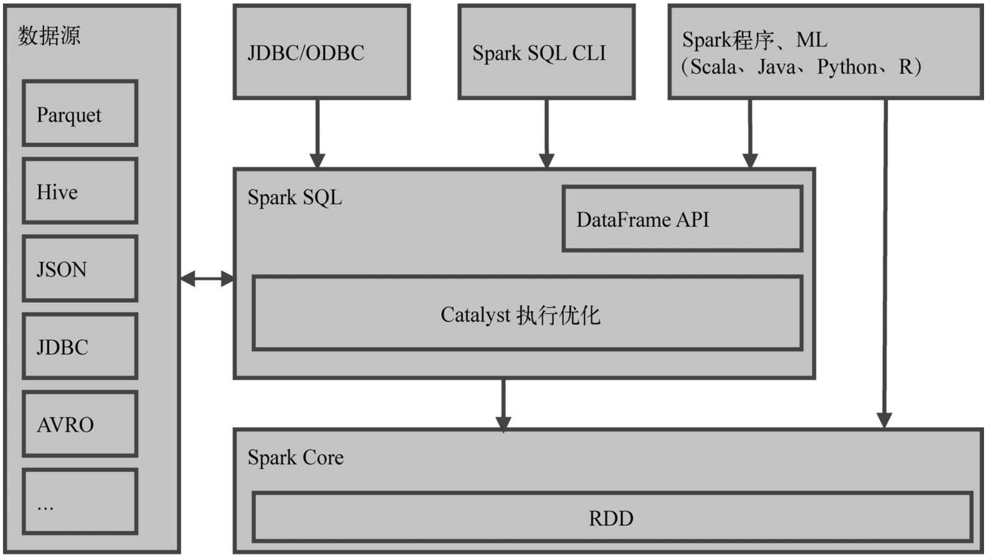


图5-3 Spark SQL的结构图

Spark SQL是Spark Core之上的一个模块，所有SQL操作最终都通过Catalyst翻译成类似普通Spark程序一样的代码，被Spark Core调度执行，其过程也有Job、Stage、Task的概念。

5.2.2 Catalyst 执行优化器

Catalyst是Spark SQL执行优化器的代号，所有Spark SQL语句最终都通过它来解析、优化，最终生成可以执行的Java字节码。因此，Catalyst是Spark SQL最核心的部分。而且不同于Shark使用Hive执行优化器的做法，Catalyst是专为Spark设计的，并结合了Scala语言本身的优势，使得性能甚至超过手工编写的Scala代码。

评判执行优化器的标准只有一条，那就是执行效率，或者说执行速度是否足够快。Spark SQL刚发布时，即与当时的Shark进行了一次对比性能测试，使用的是第三方的针对离线数据分析场景的TPC-DS基准测试，结果在多个场景下性能都显示出了绝对的优势，具体如图5-4所示。

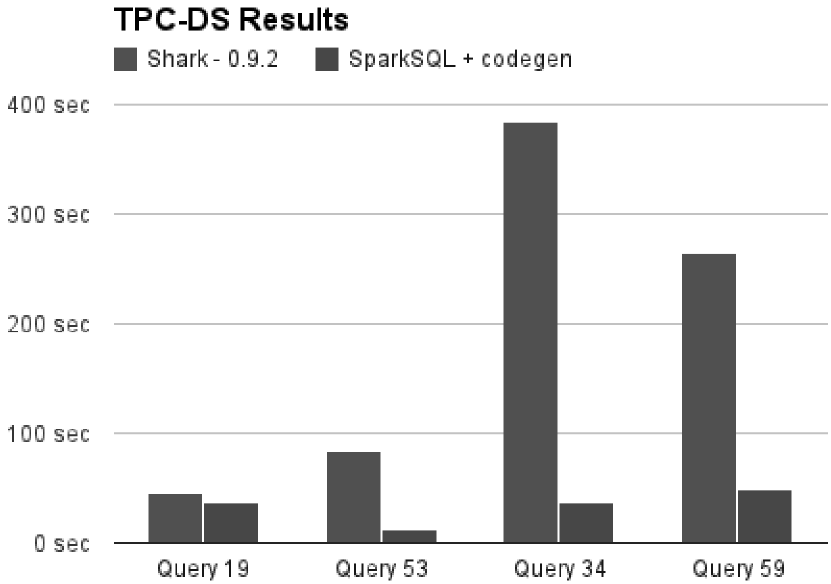


图5-4　对比性能测试结果（另见彩插图5-4）

除了性能上的优势之外，Catalyst的扩展性也非常好，它不但可以很方便地添加新的优化技术和新的特性，而且还可以很方便地扩展优化器，比如添加新的数据类型或针对特定数据源的优化规则。

Catalyst最主要的数据结构是树，所有SQL语句都会用树结构来存储，树中的每个节点有一个类（class），以及0或多个子节点。Scala中定义的新的节点类型都是`TreeNode`这个类的子类。这些对象都是不可变的。

这里以表达式`x+(1+2)`为例介绍一下，其中每个元素都对应一个类型。这个表达式总共有3个类型，具体如下所示。

- **`Literal(value: Int)`**。一个整数常量类型，`1`和`2`都是这个类型。
- **`Attribute(name: String)`**。表示一个列属性，比如`x`。
- **`Add(left: TreeNode, right: TreeNode)`**。表达式类型，用于求和。

所以，表达式最终在Scala中的定义是：

```
Add(Attribute(x), Add(Literal(1), Literal(2)))
```

这对应一棵如图5-5所示的树。

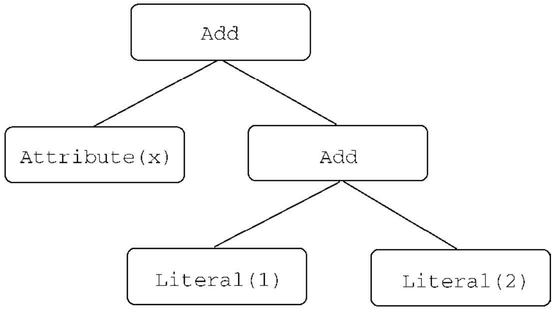


图5-5 树结构

Catalyst另外一个非常重要且基础的概念是规则。基本上，所有优化都是基于规则的。可以用规则对树进行操作，树中的节点是只读的，所以树也是只读的。规则中定义的函数可能实现从一棵树转换成一棵新树。

比如，可以定义这样一个规则，将常量相加的子树合并成一个节点：

```
tree.transform {
    case Add(Literal(c1), Literal(c2)) **=>** Literal(c1+c2)
}
```

将这个规则应用到前面的示例`x+(1+2)`表达式之后，表达式就变成了`x+3`，此时就生成了一棵新树。

这里的`case`关键字是Scala的标准模式匹配的语法，可以用来对一个对象的类型进行匹配，并且可以提取其中的值（即`c1`和`c2`）。这里的类型是一个表达式类型，用于对两个`Literal`类型的对象进行求和操作。这也是函数式编程的一大特点，任何表达式都是对象，并且属于某种类型。

Catalyst会用指定的类型对树进行匹配，自动忽略不匹配的部分，只对匹配的子树调用定义的转换操作。这样做的好处：一是规则只需要关注匹配的子树即可，不用知道全貌；二是方便扩展新的树节点类型，而无须修改现在的规则，扩展性好。

此外，也可以一次定义多条规则，而且代码非常简洁：

```
tree.transform {
    case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
    case Add(left, Literal(0)) => left
    case Add(Literal(0), right) => right
}
```

在实际的执行过程中，每条规则需要运行多次才能保证对树进行完全的转换。比如`x+(1+(2+3))`，第一次转换的结果是`x+(1+5)`，第二次转换的结果是`x+6`，这才算完成。Catalyst会循环应用规则，直到树不再发生变化为止。这样虽然每个规则都非常简单且自依赖，但最终也能产生非常巨大的效果，同时非常便于维护和调试。

5

了解了Catalyst最基本的概念树和规则之后，现在让我们来看看Catalyst的执行过程。整个优化执行器可以分为以下4个阶段，具体如图5-6所示。

- 分析阶段，分析逻辑树，解决引用。
- 逻辑优化阶段。
- 物理计划阶段，Catalyst会生成多个计划，并基于成本进行对比。
- 代码生成阶段，将查询编译成Java字节码。

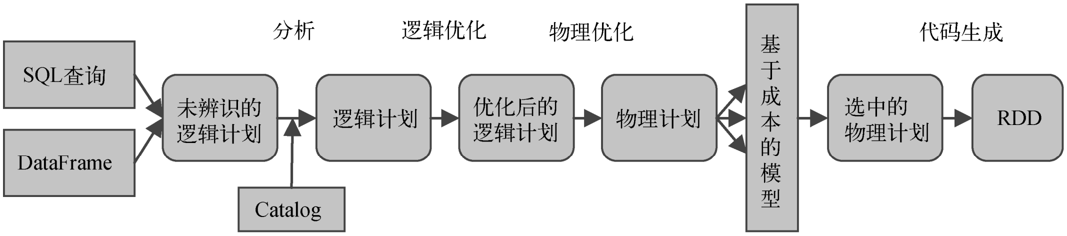


图5-6 优化执行器

这4个阶段都是基于规则的，详细分析如下。

1. 分析阶段

Spark SQL从一个关系（可以当作表来理解）的计算开始，要么来自于SQL解析后的抽象语法树（AST），或通过API调用产生的DataFrame对象。该关系中可能包含大量未辨识的属性引用或关系。比如，在SQL语句`SELECT col FROM sales`中，我们并不知道`col`的类型以及它是否是一个有效的列名，也不知道`sales`是否是表名或表的别名，直到查询所有的表定义才知道。Spark SQL使用Catalyst规则和Catalog对象来跟踪所有数据源中的表，以解决这些未辨识的属性。分析过程从创建一棵“未辨识逻辑计划”树开始，初始时树节点的类型可能是未知的，然后应用规则做如下这些事情。

- 在`catalog`对象中查找关系（表）。
- 匹配命名属性，比如列名`col`。
- 对指向相同值的属性赋予相同的ID，方便后面进一步优化，比如转换后生成的`col=col`可以优化成`True`，避免`col`的查询或计算。
- 扩散并定义类型。比如我们知道了`col`的类型，然后通过扩散就知道了`col+1`的类型，从而完成了树中所有节点的类型标识。

2. 逻辑优化阶段

在这个阶段，采用标准的基于规则的优化方法（基于成本的优化是通过应用不同的规则生成不同的计划，然后计算它们的成本）。这些规则包括常量折叠、谓词下推、投影修剪、空值传播、布尔表达式简化，等等。总之，在绝大部分情况下，都可以非常简单地添加规则。例如，当添加了固定精度的小数类型到Spark SQL时，我们希望优化聚合计算，比如求和或平均值。大家都知

道整数计算比浮点数快得多，因此在计算过程中可以使用Long类型，结果再转换成原来的固定精度的小数类型（最常用的场景在金融方面，其中两个小数位的浮点计算其实都使用整数来代替计算），整个代码不到20行，如下：

```
object DecimalAggregates extends Rule[LogicalPlan] {
    import Decimal.MAX_LONG_DIGITS
    // Double类型的最大小数位数量
    val MAX_DOUBLE_DIGITS = 15
    def apply(plan: LogicalPlan): LogicalPlan = plan
        transformAllExpressions {
        case Sum(e @ DecimalType.Expression(prec, scale)) if prec
            + 10 <= MAX_LONG_DIGITS =>
            MakeDecimal(Sum(UnscaledValue(e)), prec + 10, scale)
        // 计算平均值时，因为有一次除法，为了不降低结果的精度，
        // 对输入的精度要求高于求和
        case Average(e @ DecimalType.Expression(prec, scale))
            if prec + 4 <= MAX_DOUBLE_DIGITS =>
            Cast(
                Divide(Average(UnscaledValue(e)),
                    Literal.create(math.pow(10.0, scale), DoubleType)),
                DecimalType(prec + 4, scale + 4))
    }
}
```

3. 物理计划阶段

在物理计算阶段，Spark SQL接受一个逻辑计划作为输入，生产一个或多个物理计划，使用物理运算符来匹配Spark引擎，然后根据成本模型选择一个计划。目前，基于成本的方法只用于选择join算法：对于已知的小表，使用广播连接。因为规则可以递归遍历整个树来计算成本，所以这套机制应该也可以有更广泛的应用。

在物理计划阶段，同样也会进行基于规则的物理优化，比如将投射和过滤器用管道输送至Spark的map操作。此外，它可以将操作从逻辑计划推送至支持谓词或投影下推的数据源中进行计算。

4. 代码生成阶段

最后的代码生成阶段，将前面生成的物理计划转换成Java字节码，因为使用了Scala语言的quasiquotes特性，让这个过程变得简单且高效。quasiquotes可以让我们直接将计划树转换成抽象语法树（AST），然后Scala编译器直接生成字节码。

这里以表达式(x+y)+1为例进行介绍。如果没有代码生成，就需要针对每条记录遍历逻辑树，并解释执行每个表达式，这样会产生大量的代码分支和函数调用，效率非常低下。而使用quasiquotes之后，只需要写一些转换函数就可以生成抽象语法树，示例代码如下：

```
def compile(node: Node): AST = node match {
    case Literal(value) => q"$value"
    case Attribute(name) => q"row.get($name)"
```

```
    case Add(left, right) => q"${compile(left)} + ${compile(right)}"
}
```

表达式中的x和y对应Attribute类型，1是常量，对应Literal类型，+操作对应Add类型，q开关的字符串表示那是代码，是要被编译器编译的。

通过上面的转换之后，原始的SQL表达式(x+y)+1会翻译成Scala代码：

```
( row.get('x') + row.get('y') ) + 1
```

这基本等同于手写的效果，比解释执行至少快3倍以上。

5.3　应用场景：基于淘宝数据建立电商数据仓库

Spark SQL的分布式计算、高可扩展性、容错性、支持JDBC/ODBC的特性，完全可以作为分布式数据仓库的核心。

传统的数据仓库方案，不但成本高，而且扩展性差，所以一般选择开源的Hadoop+Hive方案即可。但Hive底层的计算框架是Hadoop MR，其性能与Spark相去甚远，而且Spark SQL基于性能更优的Spark，再加上执行器的优化，在数据仓库应用方面也会带来明显优势。比如在相同计算负载的情况下，我们只需投入部分机器即可实现Hive的效果，或者可以进行更多的计算，将统计周期从常规的天提升至小时，甚至到分钟级别。对于瞬息万变的商业环境，快速的结果查询带来的竞争力优势是不言而喻的。

通过前面章节的部署，再加上本章JDBC/ODBC Server的部署，我们已经具备了一个数据仓库的基本架构了，如图5-7所示。

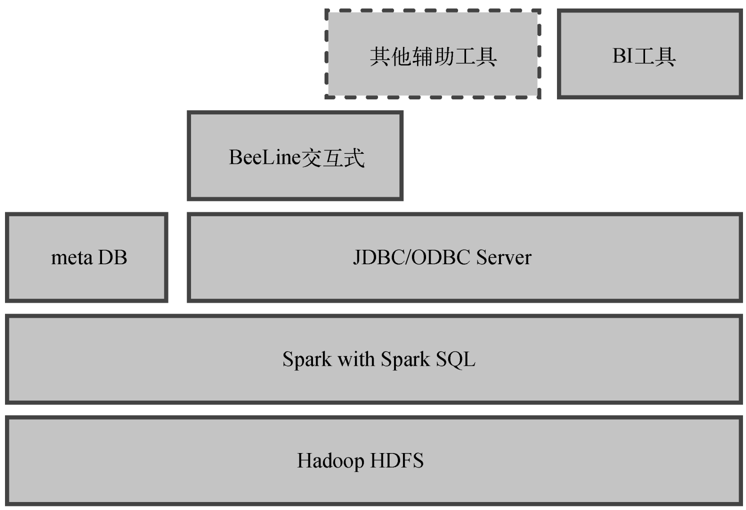


图5-7　数据仓库的架构

该架构提供了标准JDBC/ODBC接口以及交互式SQL命令行工具，可以与其他BI工具或辅助系统集成，满足企业的定制化需求。

在数据导入方面，视数据源不同，可以灵活采取多种手段，比如Hadoop HDFS客户工具、Flume、Sqoop等。

5.3.1　电商数据仓库场景

淘宝作为国内最大的电商平台，其注册商店的数量达千万级别，商品数量达十亿级别，2014年“双十一”一天的成交量达571亿元。在竞争如此激烈的平台上，作为一个商家，除了经营自己的店铺之外，还应该关注一下同行、同类商品的销售情况，做到知己知彼，进行更有针对性的营销。

作为一个卖家，可能会关注如下这些店铺信息：

- 最近7天销量最高的商家有哪些；
- 最近7天上线新品最多的店铺是谁；
- 销量增长最快的店铺是哪些。

此外，还可能关注如下这些商品信息。

- 一上市就热卖（最近一个月新上架商品中，头7天销量最高）的商品是什么。
- 行业热销宝贝排行：交易数topn的单品。
- 飙升宝贝排行榜。

现在我们尝试用公开可获取的数据，再加上Spark SQL的计算能力，来解答上面的问题。

5.3.2　数据准备和表设计

计算之前，首先要准备原始数据。由于淘宝不会公开这些数据，为了让实验过程更贴近真实，我们用简单的爬虫方式自动获取一些数据，可能不是很准确，也不完整，但作为实验应该足够了。

要回答5.3.1节里面的问题，大概需要3类数据：店铺信息、商品信息和交易流水。

为了尽可能地获取真实数据，准备数据的流程大致分为以下几步。

(1) 准备一个分类数据，用搜索引擎可以搜索到。这里我们采用3级分类，也可以手工添加其他分类数据，示例如下：

　　女装男装 女式上装 毛呢外套

　　女装男装 女式裤子 牛仔裤

　　女装男装 当季男装 长袖衬衫

　我们共准备了1000个分类。

(2) 用分类数据作为关键词，向淘宝官网搜索页面提交查询请求，读取前10页的查询结果，然后将这些商品归入当前分类中，同时读取店铺、商品信息。这样我们共获取了24万条数据。

Python爬虫的代码如下：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import sys
import urllib, urllib2, base64, json
import random
import logging, datetime, time
import StringIO, gzip
reload(sys)
sys.setdefaultencoding( "utf-8" )
# 按关键词和页面码提交HTTP搜索请求
def searchKey(keystr, page=0):
    # 发送HTTP请求
    _url="http://s.taobao.com/search?q=%s" % urllib.quote(keystr)
    if page > 0:
        _url = "%s&bcoffset=1&s=%s" % (_url, page*44)
    print _url
    _headers = {
        "Accept":"text/html,application/xhtml+xml,application/xml;
            q=0.9,image/webp,*/*;q=0.8",
        "Accept-Encoding":"gzip, deflate, sdch",
        "Accept-Language":"zh-CN,zh;q=0.8,en-US;q=0.6,en;q=0.4",
        "Cache-Control":"max-age=0",
        "Connection":"keep-alive",
        "Referer":"http://www.taobao.com",
        "User-Agent":"Mozilla/5.0 (Windows NT 6.1; WOW64)
            AppleWebKit/537.36 (KHTML, like Gecko)
            Chrome/42.0.2311.135 Safari/537.36"
    }
    # HTTP返回结果
    req = urllib2.Request(url= _url, headers= _headers)
    f = urllib2.urlopen(req, timeout=10)
    if 200 != f.getcode():
        raise Exception("create scanner fail")
    # 解压缩
    gzip_data=f.read()
    gzip_stream = StringIO.StringIO(gzip_data)
    f_gzip = gzip.GzipFile(fileobj = gzip_stream)

    # 根据关键词特征提取页面内容中的g_page_config字段
    ks = "    g_page_config = "
    klen = len(ks)
    for l in f_gzip.readlines():
        if l[:klen] == ks:
            s=l[klen : len(l)-2]
            return s
# 根据分类，构造搜索关键词进行查询
def searchClass(class_one, class_two, class_three, class_four, page):
```

```
    q_key="%s %s" % ( class_three, class_four )
    g_page_config = searchKey(q_key, page)
    cfg = json.loads(g_page_config)

    # 解析JSON
    today = datetime.datetime.now().strftime('%Y-%m-%d')
    try:
        for i in cfg['mods']["itemlist"]["data"]["auctions"]:

            detail_url = "http:" + i["detail_url"]

            # goods_base_info
            goods_id = i["nid"]
            goods_name = i["raw_title"]
            store_id = i["user_id"]
            store_name = i['nick']
            # online_time =
            print "goods_base_info %s|%s|%s|%s|%s|%s|%s" %
               (goods_id, goods_name, store_id, class_one,
               class_two, class_three, today)


            # store_base_info
            # store_id, store_name
            is_tmall = 1 if i["shopcard"]["isTmall"] else 0
            location_city = i["item_loc"]
            store_url = i["shopLink"] # 店铺链接
            print "store_base_info %s|%s|%s|%s|%s" %
                ( store_id, store_name, is_tmall, location_city, today)


            # store_credit_info
            credit_as_seller = i["shopcard"]["sellerCredit"]
            score_goods_desc = i["shopcard"]["description"][0]
            score_service_manner = i["shopcard"]["service"][0]
            score_express_speed = i["shopcard"]["delivery"][0]
            # view_price
            print "store_credit_info %s|%s|%s|%s|%s|%s" %
                (store_id, credit_as_seller, score_goods_desc,
                score_service_manner, score_express_speed, today)

            # goods_sale_info
            price = i["view_price"]
            express_price = i["view_fee"]
    except:
        pass

if __name__ == "__main__":

    if len(sys.argv) != 3:
        print "Usage: %s taobao_class_line_num download_page_num" % sys.argv[0]
        sys.exit(0)

    line_no  = int(sys.argv[1])
    page_num = int(sys.argv[2])
```

```
    f=open('tb_class_2014.utf8.txt', 'r')
    line = 0
    for l in f.readlines():
        line += 1
        if line != line_no:
            continue

        arr = l.strip().split("\t")

        p=0
        while p < page_num:
            try:
                searchClass(arr[1], arr[2], arr[3], arr[4], p)
            except:
                time.sleep(random.uniform(30,60))
            p += 1
            time.sleep(random.uniform(1,3))
```

(3) 交易流水不太好获取，所以我们用随机数的方式填充。我们生成了700万条记录。

有了数据之后，在导入之前，我们需要准备如下4张基础表。

表一：店铺基础信息。其内容如下：

```
CREATE TABLE IF NOT EXISTS store_base_info (
    store_id STRING COMMENT '店铺ID',
    store_name STRING COMMENT '店铺名',
    is_tmall TINYINT COMMENT '1:天猫, 0:淘宝',
    location_city STRING COMMENT '所在地区',
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'
STORED AS INPUTFORMAT
'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT
'org.apache.hadoop.hive.ql.io.RCFileOutputFormat'
;
```

表二：店铺信用、服务信息。其内容如下：

```
CREATE TABLE IF NOT EXISTS store_credit_info (
    store_id STRING COMMENT '店铺ID',
    credit_as_seller INT COMMENT '卖家信用',
    score_goods_desc INT COMMENT '宝贝与描述相符',
    score_service_manner INT COMMENT '卖家的服务态度',
    score_express_speed INT COMMENT '卖家发货的速度',
    info_update_date DATE COMMENT '信息最后更新日期'
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'
STORED AS INPUTFORMAT
'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT
'org.apache.hadoop.hive.ql.io.RCFileOutputFormat'
;
```

表三：商品基本信息，包含尽可能全的商品。其内容如下：

```
CREATE TABLE IF NOT EXISTS goods_base_info (
    goods_id STRING COMMENT '商品ID',
    goods_name STRING COMMENT '商品名',
    store_id STRING COMMENT '店铺ID',
    class_one STRING COMMENT '一级分类',
    class_two STRING COMMENT '二级分类',
    class_three STRING COMMENT '三级分类'
    , info_acquire_date DATE COMMENT '信息获取时间'
    , date_added DATE COMMENT '上架时间'
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'
STORED AS INPUTFORMAT
'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT
'org.apache.hadoop.hive.ql.io.RCFileOutputFormat'
;
```

表四：交易流水，数据随机填充。其内容如下：

```
CREATE TABLE IF NOT EXISTS goods_sale_info (
    goods_id STRING COMMENT '商品ID',
    data_date DATE COMMENT '数据日期',
    price INT COMMENT '价格x10',
    day_sale_count_total INT COMMENT '当天总销量（分下面3类）'
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'
STORED AS INPUTFORMAT
'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT
'org.apache.hadoop.hive.ql.io.RCFileOutputFormat'
;
```

然后就可以在beeline中使用`LOAD DATA`命令导入数据了。下面以店铺信息表为例给出代码，其他类似：

```
LOAD DATA LOCAL INPATH '/root/book/data/store_base_info.txt'
    OVERWRITE INTO TABLE store_base_info;
```

在实际数据仓库的运营过程中，数据来源可能非常多，有传统RMDB、日志文件或者其他系统，此时可以借助一些ETL工具来提升效率。此外，数据每天都会更新，需要定期导入，此时可以为每个采集周期（比如一天）创建一个分区，这样不同时间周期的数据物理存储分离，而且对于只涉及部分分区的计算，可以避免全表扫描，提升性能。

5.3.3 用 Spark SQL 来完成日常运营数据分析

现在，我们可以用Spark SQL来回答前面的问题了，只需要写SQL即可。

1. 计算最近7天销量最高的商家

相关SQL语句如下，除了销量外，这里还顺带计算了成交额和平均成交价格：

```
select
    b.store_id, c.store_name
    , sum(a.day_sale_count_total) sale_total_num
    , sum(a.day_sale_count_total * a.price) sale_total_money
    , avg(a.price) avg_price
from
    goods_sale_info a left join goods_base_info b on a.goods_id = b.goods_id
    left join store_base_info c on b.store_id = c.store_id
where
    a.data_date >= date_sub(to_date(from_unixtime(unix_timestamp())),7)
group by
    b.store_id, c.store_name
    order by sale_total_num desc
limit 10;
```

beeline返回的结果如图5-8所示。

store_id	store_name	sale_total_num	sale_total_money	avg_price
1071349722	冬珊旗舰店	259579620	141166785258	546.2039909015715
331313540	wuaiwujia168	26998924	15689525576	571.4081346423562
766395193	c546507079	26604550	13479141620	505.0960196189398
1596202434	聚心媛	24935784	12122871480	488.04140587385655
906780879	tb5347_45	22619652	12355733952	548.6956636700158
1129326215	天猫超市华南站	12674562	7568081706	594.4660937132094
2389198856	小宇宙200301	9222353	5618972239	607.6572580645161
1086392022	人生最佳时	7285230	3997397121	545.0721183480583
614528334	丽雅月77	7236780	3688330830	515.1268817204301
35243780	zxh0819	6403720	3324286855	521.9758064516129

10 rows selected (12.825 seconds)

图5-8　最近7天销量最高的商家

2. 计算最近7天上线新品最多的10家店铺

相关SQL语句如下：

```
select
    a.store_id, b.store_name
    , count(distinct(a.goods_id)) new_goods_num
from
    goods_base_info a left join store_base_info b on a.store_id = b.store_id
where
    a.date_added >= date_sub(to_date(from_unixtime(unix_timestamp())),7)
    and a.date_added < to_date(from_unixtime(unix_timestamp()))
group by
    a.store_id, b.store_name
order by new_goods_num desc
limit 10;
```

执行结果如图5-9所示。

store_id	store_name	new_goods_num
1071349722	冬珊旗舰店	312
1129326215	天猫超市华南站	247
766395193	c546507079	171
1086392022	人生最佳时	157
906780879	tb5347_45	122
90925757	李宁古星专卖店	96
859515618	博库图书专营店	94
407700539	周大福官方旗舰店	93
703040105	远达运动专营店	92
104847814	天海蓝蓝d	91

图5-9　最近7天上线新品最多的10家店铺

3. 计算最近7天相对之前7天销量增长最快的店铺

相关SQL语句如下：

```
select
    b.store_id, c.store_name
    , sum(
        case
            when a.data_date >= date_sub(to_date(from_unixtime(unix_timestamp())),7)
                then
                -- 最近一个周期
                a.day_sale_count_total
            else
                -- 减去前一个周期
                0-a.day_sale_count_total
        end
    ) sale_total_inc
from
    goods_sale_info a
    left join goods_base_info b on a.goods_id = b.goods_id
    left join store_base_info c on b.store_id = c.store_id
where
    -- 取最近两个周期
    a.data_date >= date_sub(to_date(from_unixtime(unix_timestamp())),7+7)
group by
    b.store_id, c.store_name
order by sale_total_inc desc
limit 10;
```

执行结果如图5-10所示。

store_id	store_name	sale_total_inc
1071349722	冬珊旗舰店	102210552
331313540	wuaiwujia168	10474358
766395193	c546507079	10322890
1596202434	聚心媛	9429720
906780879	tb5347_45	8915328
1129326215	天猫超市华南站	4851912
2389198856	小宇宙200301	3778948
1086392022	人生最佳时	2842071
614528334	丽雅月77	2816130
1820998937	林氏家装服务商	2460960

10 rows selected (6.812 seconds)

图5-10　最近7天相对之前7天销量增长最快的店铺

4. 计算一上市就热卖的商品

下面的SQL代码用于计算在最近一个月新上架的商品中，头7天销量最高的商品：

```
select
    b.goods_id
    , sum(a.day_sale_count_total) sale_total_num
    , b.goods_name
from
    goods_sale_info a left join goods_base_info b on a.goods_id = b.goods_id
where
    a.data_date  >= date_sub(to_date(from_unixtime(unix_timestamp())), 30)
    and b.date_added >= date_sub(to_date(from_unixtime(unix_timestamp())), 30)
    and a.data_date >= b.date_added and a.data_date <= date_add(b.date_added, 7)
group by b.goods_id, b.goods_name
order by sale_total_num desc
limit 10;
```

执行结果如图5-11所示。

goods_id	sale_total_num	goods_name
41880188438	42814	回头客铜锣烧 红豆味1500g蛋糕整箱西式糕
38604005707	41485	2015春夏女鞋真皮坡跟鱼嘴鞋女士凉鞋大码
44280463997	40603	欧洲站2015女鞋夏春新款 真皮镂空鱼嘴凉鞋
44282081027	39179	【天天特价】现货山东烟台大樱桃车厘子 新
45085732422	38358	夏季韩版女鞋鱼嘴真皮厚底菱形镂空单鞋低
45096564441	37527	达芙妮正品2015夏季新款粗跟中跟女凉鞋真
44560391138	37204	2015夏季包头凉鞋女式新版尖头真皮水钻电
44783563031	36889	单鞋平底鞋凉鞋女夏时尚平跟夹趾水钻凉拖
44872412802	36780	2015春夏新款凉鞋女中跟厚底平底凉拖坡跟
44277915317	36629	【天天特价】尖头镂空一字搭扣真皮女单鞋

10 rows selected (4.67 seconds)

图5-11　最近一个月新上架的商品中头7天销量最高的商品

5. 计算分类交易量最高的10个商品

下面的SQL语句用于计算分类“女装男装”→“女式上装”下最近3天成交额top10的商品：

```
select
    a.goods_id
    , sum(a.day_sale_count_total) sale_total
    , b.goods_name
from
    goods_sale_info a
    left join goods_base_info b on a.goods_id = b.goods_id
where
    a.data_date >= '2015-05-02' and a.data_date <= '2015-05-04'
    and b.class_one = "女装男装" and b.class_two = "女式上装"
group by
    a.goods_id, b.goods_name
order by sale_total desc
limit 10;
```

执行结果如图5-12所示。

goods_id	sale_total	goods_name
44676289773	9066	羊绒衫女春款开衫打底毛衣针织毛呢外套羊
41564861087	8606	2014秋装外套拉古毛针织开衫毛衣中长款羊
41091078400	8578	欧美范厚貂绒毛呢西服领外套长款毛衣针织
42202669403	8550	长袖毛呢拼接不对称加厚针织毛衣外套日系
40764749010	8232	柳萍14秋冬新款女士中长款貂绒衫 大翻领
41144468170	7864	2014秋冬新品欧洲站羊毛呢大衣 时尚简约
42598641497	7850	韩版女装毛衣阿里巴巴批发1688官网中长款
44095350465	7702	春季新品藏绒多斯女士桑蚕丝薄开衫宽松大
41741531906	7688	新款 韩版复古斗篷显瘦蝙蝠袖毛衣针织
40554569655	7196	秋装新风尚V领羊绒开衫 女修身针织外套

10 rows selected (3.107 seconds)

图5-12 “女装男装”→“女式上装”下最近3天成交额top10的商品

6. 计算最近7天，相对之前7天，销量增长最多的商品

相关SQL语句如下：

```
select
    a.goods_id, b.goods_name
    , sum(
        case
            when a.data_date >= date_sub(to_date(from_unixtime(unix_timestamp())),7)
                then
                -- 最近一个周期
                a.day_sale_count_total
            else
                -- 减去前一个周期
                0-a.day_sale_count_total
```

```
        end
    ) sale_total_inc
from
    goods_sale_info a left join goods_base_info b on a.goods_id = b.goods_id
        where
    -- 取最近两个周期
    a.data_date >= date_sub(to_date(from_unixtime(unix_timestamp())),7+7)
group by
    a.goods_id, b.goods_name
order by sale_total_inc desc
limit 10;
```

执行结果如图5-13所示。



goods_id	sale_total_inc	goods_name
44933443899	71136	2015春夏新款罗马鞋厚底女单鞋真皮鱼嘴
41880188438	70569	回头客铜锣烧 红豆味1500g蛋糕整箱西式
38604005707	70050	2015春夏女鞋真皮坡跟鱼嘴鞋女士凉鞋大
44930245590	68226	日系复古街头民族风星星短裤小清新抽绳
44812902397	66594	达派双肩包正品腰包休闲潮旅行两用多功
37951211534	63486	莱卡金顿 夏季新款坡跟凉拖 女高跟凉鞋
44949312591	62832	2015年春夏新款红蜻蜓鱼嘴凉鞋女真皮高
44139567572	62049	2015春夏季新款丽人达芙妮真皮女鞋单鞋
38348298688	61716	欧美夏季新款尖头牛皮漆皮T型镂空全真皮
44223755277	60459	夏季新款女士网纱凉鞋中老年妈妈鞋网面

10 rows selected (6.052 seconds)

图5-13　最近7天相对之前7天，销量增长最多的商品

5.3.4　Spark SQL 在大规模数据下的性能表现

在5.3.3节，我们基本实现了一个数据仓库雏形，解决了电商运营的6个问题。

当然，基于MySQL等解决方案，通过分库分表的形式，我们也能完成相应的任务，但Spark SQL的优势在于当数据流扩大以后，其性能只是线性增加的，而且可以通过平行扩展来完成扩容，而不用涉及数据迁移等问题。

为了观察Spark的性能表现与数据大小的关系，我们尝试将数据量放大（简单的重复复制）到原来的10倍、100倍和1000倍，来看看Spark的性能表现。表5-1显示的是原始数据的大小。

表5-1　原始数据的大小

数据文件大小	表　　名
220 MB	goods_sale_info
40 MB	goods_base_info
7 MB	store_base_info
5 MB	store_credit_info

这里以场景一“计算最近7天销量最高的商家”为例进行测试，测试使用的机器都比较旧而且配置一般，只有6个计算节点（存储节点共用机器），每台机器8核CPU，可用内存8 GB，这些机器在同一个机房，使用普通千兆网卡进行通信。表5-2是数据放大之后的运行时间。

表5-2 数据放大之后的运行时间

测试数据的倍数	数据量	用时（秒）	用时相对增长倍数	总增长倍数
1	0.2 GB × 0.04 GB	10		
10	2 GB × 0.4 GB	56	5.6	5.6
100	20 GB × 4 GB	550	9.8	55
1000	200 GB × 40 GB	5635	10.2	564

即便数据增长至1000倍，SQL中JOIN操作的左右两张表数据都增长1000倍，总的用时的增长还是与数据增长的倍数持平，而且虽然在1000倍时数据量已经超过可用内存总量的4倍，但计算依然可以继续并保持线性。因此，可以得出结论：Spark的性能基本上与数量大小保持线性关系，这样的好处就非常明显，使得未来集群的计算能力的扩充变得非常简单，只需要增加计算节点即可。

第 6 章

Spark流式计算

6

前面提到的Spark计算或Spark SQL计算，它们类似的地方是涉及的数据量庞大，计算时间长，典型场景下一次计算的耗时一般是数分钟或者数小时。但在实际业务场景中，还有一类称作流式计算的应用，需要实时对大量数据进行快速处理，最明显的特点就是处理周期短，一般是分钟级，甚至秒级、毫秒级，而且是7 × 24小时连续不断地进行计算。

对于实时流式数据计算，Spark通过Spark Streaming组件提供了支持。Spark Streaming基于Spark核心，具备可扩展性、高吞吐量、自动容错等特性，数据来源支持Kafka、Flume、Twitter、ZeroMQ、Kinesis或TCP socket等，处理时可以使用map、reduce、join、window等高级函数来实现复杂逻辑，结果可以写入文件系统、数据库或其他实时展示系统。而且由于Spark平台的高度整合性，我们还可以使用前面提到的Spark SQL、DataFrame，以及后面即将介绍的机器学习、图计算等Spark高级算法。图6-1显示了Spark Streaming计算的基本过程。

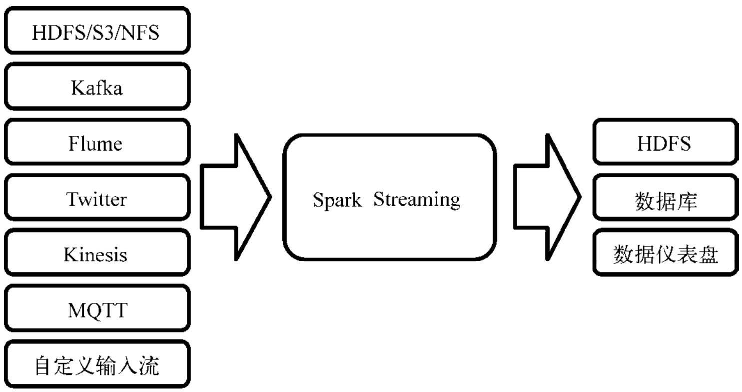


图6-1 Spark Streaming计算过程

在内部，Spark Streaming接收实时数据，按周期将数据分成多批次（batch），按批次提交给Spark核心来调度计算，如图6-2所示。

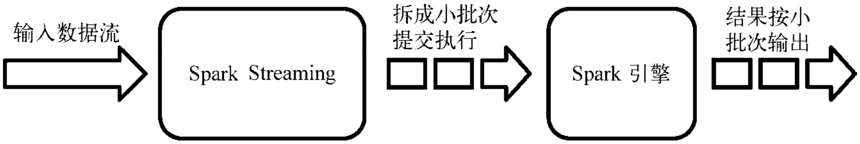


图6-2 Spark Streaming按批次执行

Spark Streaming使用的数据抽象是`DStream`（discretized stream），它表示连续的数据流，但内部其实是通过RDD序列来存储的。

Spark Streaming计算支持Scala、Java、Python语言，暂时不支持R。

6.1 Spark Streaming 基础知识

本节通过一个简单示例介绍如何开发Spark流式计算程序。

6.1.1 入门简单示例

在介绍完整的Spark流式计算之前，我们先给出一个最简单的示例：Spark Streaming版本的WordCount。使用Linux下的终端工具netcat（简称nc）启动一个TCP服务，用户手工输入文本，Spark通过tcp socket读取文本、统计词频并输出，基本过程如图6-3所示。

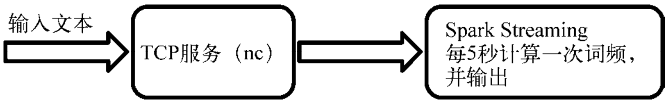


图6-3 Spark Streaming简单示例

先启动TCP socket，在Linux下通过nc启动，端口为9999：

```
nc -lk 9999
```

然后启动spark-shell：

```
./bin/spark-shell
```

在spark-shell中输入如下代码，运行一个Spark Streaming版WordCount程序：

```
// 导入类
import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds, StreamingContext}

// 创建StreamingContext，基于shell下默认的SparkContext
val ssc = new StreamingContext(sc, Seconds(5))

// 创建DStream，这是Streaming计算下的RDD
val lines = ssc.socketTextStream("localhost", 9999)
```

6

```
// 基本的操作函数与RDD同名
val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)

// 打印结果到标准输出，只打印DStream中每个RDD的前10个元素
// 注意，print不同于RDD中的Action操作，不会触发真正的调度执行
wordCounts.print()

// 这里才正式启动计算
ssc.start()

// 等待执行结束（出错或Ctrl-C退出）
ssc.awaitTermination()
```

现在，我们可以在前面启动的nc客户端中输入文本了：

```
hello world !!!
```

回到spark-shell界面，我们可以看到这样的输出：

```
(hello,1)
(world,1)
(!!!,1)
```

6.1.2　基本概念

1. StreamingContext

StreamingContext是Spark Streaming编程的最基本环境对象，就像Spark编程中的SparkContext一样。StreamingContext提供最基本的功能入口，包括从各途径创建最基本的对象DStream（就像Spark编程中的RDD）。

创建StreamingContext的方法很简单，生成一个SparkConf实例，设置程序名，指定运行周期（示例中是5秒），这样就可以了：

```
val conf = new SparkConf().setAppName("SparkStreamingWordCount")
// 注：spark-shell下，需要先调用sc.stop()来停止默认的SparkContext，
// 因为默认同时只能运行一个SparkContext
val ssc = new StreamingContext(conf, Seconds(5))
```

运行周期为5秒，表示流式计算每间隔5秒执行一次。这个时间的设置需要综合考虑程序的延时需求和集群的工作负载，应该大于每次的运行时间。

StreamingContext还可以从一个现存的org.apache.spark.SparkContext创建而来，并保持关联，比如上面示例中的创建方法：

```
// sc是一个已经存在的SparkContext
val ssc = new StreamingContext(sc, Seconds(5))
```

后面可以通过ssc.sparkContext来获取这个对象。

StreamingContext创建好之后，还需要下面这几步来实现一个完整的Spark流式计算：

(1) 创建一个输入DStream，用于接收数据；

(2) 使用作用于DStream上的Transformation和Output操作来定义流式计算（Spark程序是使用Transformation和Action操作）；

(3) 启动计算，使用streamingContext.start()；

(4) 等待计算结束（人为或错误），使用streamingContext.awaitTermination()；

(5) 也可以手工结束计算，使用streamingContext.stop()。

2. DStream抽象

DStream（discretized stream）是Spark Streaming的核心抽象，类似于RDD在Spark编程中的地位。DStream表示连续的数据流，要么是从数据源接收到的输入数据流，要求是经过计算产生的新数据流。DStream的内部是一个RDD序列，每个RDD对应一个计算周期。比如，在上面的WordCount示例中，每5秒一个周期，那么每5秒的数据都分别对应一个RDD，如图6-4所示，图中的时间点1、2、3、4代表连续的时间周期。

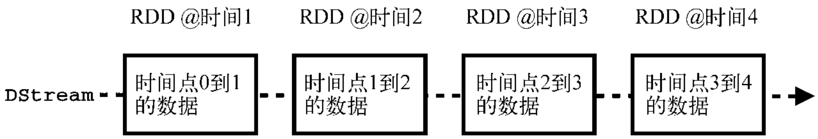


图6-4　DStream内部是RDD序列

所有应用在DStream上的操作，都会被映射为对DStream内部的RDD上的操作，比如上面的WordCount示例中对lines DStream的flatMap操作，会被映射到lines内部的RDD上，如图6-5所示。

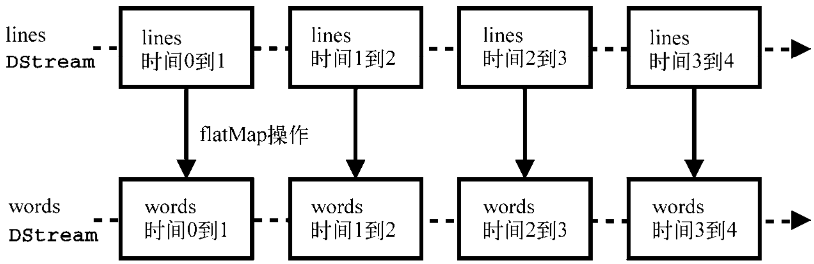


图6-5　DStream应用flatMap操作

RDD操作将由Spark核心来调度执行，但DStream屏蔽了这些细节，给开发者更简洁的编程体验。当然，我们也可以直接对DStream内部的RDD进行操作（后面会讲到）。

3. 输入DStream

输入DStream是DStream的起始，程序中的第一个DStream用于从各类数据源接收数据，比如上面WordCount示例中的lines就是一个输入DStream，从netcat TCP Socket中读取数据。而且，输入DStream中包含一个接收器对象，用于读数据并缓存在Spark内存中来等待进一步处理。

Spark Streaming支持3种输入DStream，如下。

- **基本型**。在StreamingContext的API中直接提供，包括文件系统、Socket、Akka actor等类型。
- **高级型**。需要通过额外的库来实现，比如Kafka、Flume、Kinesis、Twitter，而且在编译时需要设置依赖，并打包进来。
- **自定义型**。完全由用户自行指定并编码实现。

当然，同时使用多个输入DStream也是可以的，创建它们就好了，只是要保证分配的core的数量要大于输入DStream的数量。

● 基本输入DStream

前面示例中的socketTextStream就是一种基本型的输入DStream。此外，我们还可以用fileStream生成从文件或Akka actor读取数据的输入DStream。

从文件生成输入DStream的最简单方法是使用方法textFileStream，接收一个HDFS文件系统兼容的目录作为参数，Spark会实时监控这个目录，以文件为单位。新出现的文件会被按照文本文件格式读取到Spark内存中。对于文件的几点要求如下：

- 目录中的所有文件的格式相同；
- 只处理目录下的文件，不包括子目录下的文件；
- 文件进入目录的方式必须是原子的，比如被从其他目录移动或重命名得来；
- 文件进入目录之后，建议不要再被修改，因为只会被读取一次，新追加的内容不会被读取。

文件型输入DStream的完整调用方式如下（注：Python不支持下面的形式，只支持textFileStream）：

```
streamingContext.fileStream[KeyClass, ValueClass, InputFormatClass](dataDirectory)
```

● 高级输入DStream

高级类型的输入Dstream并不由Spark提供，而是由其他项目提供，需要在编译时包含进来。比如，若你想使用Kafka，需要在编译时引用spark-streaming-kafka_2.10，下面是使用示例，详细信息可以参考Kafka API文档：

```
import org.apache.kafka.clients.producer.{ProducerConfig, KafkaProducer,
    ProducerRecord}
// zkQuorum
// group
```

```
//
// zkQuorum : ZooKeeper服务器列表
// group    : Kafka消费者组名
// topics   : 链表，包含一个或多个Kafka主题，用于消费
val topicMap = topics.split(",").map((_, numThreads.toInt)).toMap
val lines = KafkaUtils.createStream(ssc, zkQuorum, group, topicMap).map(_._2)
```

当然，除了编译时的链接和编程，我们还需要运行一个Kafka集群用于提供数据源，所以外部高级数据源的使用都需要这3步：

(1) 有可用的外部系统提供数据源；

(2) 编程使用外部输入DStream；

(3) 编译时链接外部库。

- **接收可靠性**

在使用数据源时，我们还需要留意一下可靠性。有一类数据源本身就是可靠的，比如Kafka和Flume。它们对于传输的数据有确认机制，如果我们能够利用好这一机制，恰当地对收到的数据进行确认，那么可以确保数据在任何异常情况下都不会丢失。

4. DStream操作

DStream操作分两类。

- **Transformation操作**。类似于RDD的Transformation，对一个DStream进行计算，输出新的DStream。
- **output操作**。类似于RDD的Action操作，将DStream输出至外部系统，比如文件系统、数据库或BI报表系统。

- **DStream Transformation操作**

DStream支持的Transformation操作有map、flatMap、filter、repartition、union、count、reduce、countByValue、reduceByKey、join、cogroup、transform、updateStateByKey，大部分的使用方法与RDD的同名方法类似，几个比较特殊的见下面的详细讲解。

- **transform**

DStream内部其实是RDD序列，transform提供了直接操作DStream内部RDD的方法，对于那些DStream没有提供的RDD操作，可以通过transform调用，非常灵活。比如DStream没有提供与其他RDD进行join操作的方法，通过transform可以实现：

```
// rdd2是一个已经存在的RDD
val joinedDStream = d.transform(rdd => {
    rdd.join(rdd2)
})
- UpdateStateByKey
```

6

由于流式计算是连续进行的，很大可能都是7×24小时不间断的，因此可能需要保存一些状态。简单的全局广播变量显然是不够的，所以Spark Streaming提供了专门的状态更新方法`updateStateByKey`，比如我们可以改造一下前面的WordCount示例，不仅统计每次新接收到的数据，而且让词频数据可以累加。先定义一个状态更新函数：

```
def updateFunction(newValues: Seq[Int], runningCount: Option[Int]): Option[Int] = {
    val preCount = runningCount.getOrElse(0) // 历史累计值
    val newCount = newValues.sum // 当前周期新数据的值
    Some(newCount + preCount) // 再次累加
}
```

然后就可以调用了；不需要初始值，因为`updateFunction`里已经处理了（看到`getOrElse(0)`了吗 :)）：

```
val runningCounts = pairs.updateStateByKey[Int](updateFunction _)
```

注意　使用`updateStateByKey`必须配置检查点（checkpoint）目录，否则会报错，详情请参考6.2.2节。

- **窗口（window）操作**

流式计算是周期性进行的，有时除了处理当前周期的数据，还需要处理最近几个周期的数据，这时就需要窗口操作方法了。我们可以设置数据的滑动窗口，将数个原始Dstream合并成一个窗口DStream，如图6-6所示。

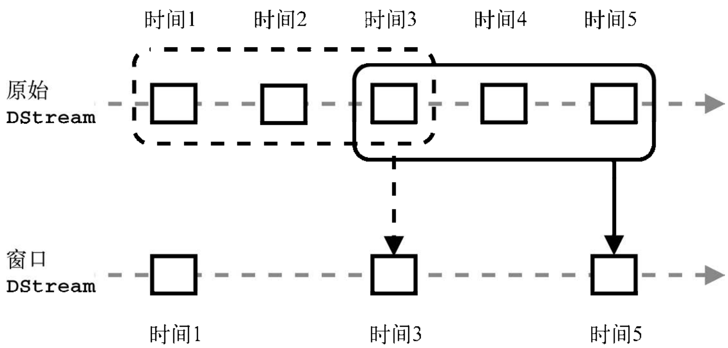


图6-6　窗口DStream

窗口通过如下两个参数来确定。

- **窗口长度**。即窗口跨越的周期次数，上图示例中是3。
- **滑动区间**。从当前窗口到下一个窗口间隔的周期数量，上图示例中是2。

如果这两个参数的值都是1，那么效果跟不使用窗口一样。每个窗口内的所有周期数据都会

合并成一个RDD保存在DStream中，上图中每次窗口处理3个周期的数据，每次滑过2个周期，而且计算周期也比原始值扩大了一倍。

我们可以使用窗口函数（比如windowLength、slideInterval）来生成一个带窗口的DStream，然后当作普通DStream来用。另外也可以直接使用加窗口参数的专用方法，常用的有countByWindow、reduceByWindow、reduceByKeyAndWindow、reduceByKeyAndWindow、countByValueAndWindow。

- **Output操作**

Output操作将DStream结果输出至外部系统，这也意味着一段DStream计算的结束。目前可用的Output操作如下。

- **print()**。调试测试用，打印DStream中各RDD的头部10个成员，print也可以添加一个参数指定输出的头部成员数量，比如print(30)。
- **saveAsTextFiles(prefix, [suffix])**。保存为文本文件、本地磁盘系统或HDFS，文件名格式是“prefix-TIME_IN_MS[.suffix]”。
- **saveAsObjectFiles(prefix, [suffix])**。保存为SequenceFile格式，文件名格式同上。
- **foreachRDD(func)**。最灵活的Output操作，传入一个函数，直接作用在RDD上。传入的函数可以实现任意功能，比如写文件、写DB，或者通过网络发送出去。需要注意的是，func是在调用Spark流式计算程序的Driver进程中执行的，但func中一般会调用RDD的Action操作，那却是在worker中执行的。

6.1.3 高级操作

1. 缓存与持久化

DStream也可以像RDD一样非常简单地缓存到内存中，我们统一称作**持久化**，对应的级别是内存。调用persist()可以把DStream及包含的RDD全部缓存到内存中，这对于那些需要被反复使用的DStream尤其重要。窗口函数和updateStateByKey默认会自动持久化，不需要调用persist()，因为数据的确会被多次使用。

对于从网络接收数据的输入DStream（比如Kafka、Flume、sockets等），默认的持久化级别是复制数据到两个节点上，以确保容错能力。

2. 打包、发布和监控

- **编译链接**

使用Spark组件的程序，在编译打包时都需要配置依赖，SparkStreaming也不例外。

SBT的配置方法：

```
libraryDependencies += "org.apache.spark" % "spark-streaming_2.10" % "1.4.1"
```

6

Maven的配置是：

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming_2.10</artifactId>
    <version>1.4.1</version>
</dependency>
```

如果使用了其他类型的输入数据源，也需要包含进来，常用的数据源对应的包如下。

- Flume：spark-streaming-flume_2.10。
- Kafka：spark-streaming-kafka_2.10。
- MQTT：spark-streaming-mqtt_2.10。
- Twitter：spark-streaming-twitter_2.10。
- ZeroMQ：spark-streaming-zeromq_2.10。

如果想查看Spark支持的所有第三方包，可以在http://search.maven.org/上输入g:"org.apache.spark" AND v:"1.4.1"进行搜索。

- **部署**

为了让Spark流式计算程序在集群上也能正常工作，还有一些额外事项需要注意。

- 在打包JAR包时，我们需要提供Spark自身之外的其他所有第三方jar包。比如若使用了Kafka，不但要提供spark-streaming-kafka_2.11-1.4.1.jar，还要提供它依赖的其他Spark本身不提供的包。解决方法：可在用`spark-submit`提交时使用`--jars`选项附带上这些依赖的包，或者在编译时使用静态编译的方式，将所有依赖的包全部打包进去。
- 配置足够的内存资源，因为每个周期接收到的数据都会缓存在内存中，而且如果使用了窗口函数，需要缓存的会更多。
- 配置检查点时，需要一个HDFS API兼容的文件系统目录（比如Hadoop HDFS、S3等），请确保有相应的系统可以使用。有些操作要求必须配置检查点，比如`updateStateByKey`。
- 配置Driver程序自动重启。流式计算一般都是一直运行的，而Driver程序是Spark程序的中枢，只运行在某一个节点上。Driver程序本身异常退出或者异常，都可能导致流式计算中断。每种集群模式都提供这种功能，Standalone模式下提交时可以使用`--deploy-mode cluster --supervise`选项，YARN下选择`yarn-cluster`，Mesos下可以通过Marathon来协助实现。

Spark从1.2版开始，支持WAL（Write Ahead Logs）。WAL是一种避免数据丢失的方法，开启之后，所有收到的数据在处理之前都会先写到检查点目录下，这样可以确保在Driver恢复期间数据不丢失。设置`spark.streaming.receiver.writeAheadLog.enable`为`true`可以开启此功能，但代价是降低了数据接收的吞吐量，不过可以采用并发接收的方式来降低影响。此外，如果

开启了WAL，接收数据时的复制机制可以关闭了，因为两者的目标是相同的；关闭复制的方法是设置输入DStream的存储级别为`StorageLevel.MEMORY_AND_DISK_SER`。

- **更新程序代码**

流式计算一般都长期运行，但偶尔也会更新一下计算逻辑。虽然更新程度一般也就不到一分钟的时间，但是会重启程序，如果想要在重启过程中不丢失数据，需要做一些特殊处理，我们有如下两种办法。

- ❑ **新旧程序同时运行**。先不要停止旧的程序，直接启动新的程序，待新的程序运行之后，再停止旧的程序。这种方式要求数据源支持同时向新、旧两个版本的程序发数据，而且程序上也要考虑这种特殊场景。
- ❑ **先停止旧的程序再启动新的程序**。但要保证旧的程序是被优雅地关闭的，确保关闭之前接收到的所有数据都已经处理完毕了，然后再启动新的程序，接着继续处理。这种方式只适用于支持缓存数据功能的数据源，比如Kafka、Flume，这样程序中断这段时间的数据可以临时缓存一下，待新程序启动后继续发送。还有个检查点信息的问题，启动新的程序时，在读取前一个程序的检查点信息时可能出错，因为检查点中有一些对象的序列化数据，新的对象结构可能已经改变了；解决的办法是要么用新的检查点目录，要么是删除旧的检查点目录下的所有内容。

- **监控流式计算程序运行**

Spark为每个运行中的程序都提供了一个Web界面，用于监控程序的运行，Spark Streaming程序运行时，界面上会多出一个名为Streaming标签页，专用于流式计算，如图6-7所示。



图6-7　Spark Web界面增加的一个Streaming标签

我们需要重点关注两个数据：Scheduling Delay和Processing Time（如图6-8所示）。这两者加起来就是Spark Streaming一次周期计算的总时间。



图6-8　Streaming计算的时间信息

如果这两个时间在持续增加,或者两者之和超过了创建StreamingContext时设置的周期间隔时间，那很可能是集群资源紧张了，可以增加周期，或者优化集群或程序。

6.2　深入理解 Spark Streaming

6.2.1　DStream 的两类操作

前面我们提到，DStream内部其实是RDD序列，所有的DStream操作最终都转换为RDD操作。通过分析源码，我们可以进一步窥视这种转换是如何进行的。

DStream有一些与RDD类似的基础属性：

- 依赖的其他DStream列表；
- 生成RDD的时间间隔；
- 一个名为compute的计算函数，用于生成RDD，类似于RDD的compute。

DStream的操作分为两类，一类是Transformation操作，对应RDD的Transformation操作。以flatMap为例，DStream中的flatMap不过是返回了一个新的DStream派生类FlatMappedDStream，这一点跟RDD的flatMap非常类似。DStream的flatMap定义如下：

```
/**
 * 将输入的函数作用到DStream的所有成员，并展开返回值，返回一个新的DStream
 */
def **flatMap**[U: ClassTag](flatMapFunc: T => Traversable[U]): DStream[U] =
    ssc.withScope {
    **new FlatMappedDStream**(this, context.sparkContext.clean(flatMapFunc))
}
```

而FlatMappedDStream的实现也很简单，只有十几行代码，主要作用是像RDD一样维护计算关系链，完整定义如下：

```
private[streaming]
class **FlatMappedDStream**[T: ClassTag, U: ClassTag](
    parent: DStream[T],
    flatMapFunc: T => Traversable[U]
    ) extends DStream[U](parent.ssc) {

    override def dependencies: List[DStream[_]] = List(parent)

    override def slideDuration: Duration = parent.slideDuration

    override def compute(validTime: Time): Option[RDD[U]] = {
        parent.getOrCompute(validTime).map(_.flatMap(flatMapFunc))
    }
}
```

其中compute调用DStream的getOrCompute方法用于读取RDD的内存，要么放到缓存中，要么调用接口函数compute计算生成。

DStream另外一类操作是Output操作，Output操作才会触发DStream的实际执行，作用非常类似于RDD的Action操作，那么这两者之间是什么关系呢？我们来看一下最简单的print操作，定义如下：

```
/**
 * 打印DStream内每个RDD的前面num个成员
 * 此外，输出类型的操作需要标记一下（最后的register调用），以确保里面的RDD都会被生成
 */
def **print**(num: Int): Unit = ssc.withScope {
    def foreachFunc: (RDD[T], Time) => Unit = {
        (rdd: RDD[T], time: Time) => {
            val firstNum = rdd.**take**(num + 1)
            println("-------------------------------------------")
            println("Time: " + time)
            println("-------------------------------------------")
            firstNum.**take**(num).foreach(println)
            if (firstNum.length > num) println("...")
            println()
        }
    }
    new ForEachDStream(this, context.sparkContext.clean(foreachFunc)).register()
}
```

我们看到DStream.print调用了RDD.take方法，而后者正是一个Action操作，是不是所有的Dstream输出操作最后都会调用一个RDD的Action操作呢？我们再看看saveAsTextFile和saveAsObjectFiles，它们没有直接调用RDDaction操作，但是通过foreachRDD来实现，传入的函数中调用了RDD的Action。saveAsTextFile的定义如下：

```
/**
  * 将DStream中的所有RDD保存为文本文件，RDD成员转换为相应的字符串形式
  * 各周期使用不同的文件名，格式是：prefix参数指定的前缀-以毫秒为单位的时间.suffix参数指定的后缀
  */
def saveAsTextFiles(prefix: String, suffix: String = ""): Unit = ssc.withScope {
    val saveFunc = (rdd: RDD[T], time: Time) => {
        val file = rddToFileName(prefix, suffix, time)
            **rdd**.**saveAsTextFile**(file)
    }
    this.foreachRDD(saveFunc)
}
```

相比之下，另外一个最灵活的Output操作foreachRDD完全依赖传入的函数来实现功能，所以对于foreachRDD的使用就有要求了：至少包含一个RDD Action调用。因为Spark Streaming的调度是由Output方法触发的，每个周期调用一次所有定义的Output方法，Output内部再调用RDD Action最终完成计算，否则程序只是接收数据，然后丢弃，不会执行计算。

6.2.2　容错处理

1. 背景知识

在讨论Spark Streaming容错体系之前，我们先重温一下背景知识，即DStream所依赖的RDD容错机制：RDD是只读的、可重复计算的分布式数据集，它用线性链条的形式记录了RDD从创建开始的中间每一步的计算过程，错误恢复的过程就是重新计算的过程，如果RDD某个分区因为任何原因数据丢失了，都可以使用记录起来的计算过程重新计算从而恢复数据，只要计算过程确定，即便集群出错也能保证确定的计算结果。

Spark操作的数据一般存储在有容错功能的文件系统（比如HDFS、S3）上，从这些系统上的数据生成的RDD也具有容错能力，但是这个不适用于Spark Streaming。因为大部分场景下，Spark Streaming的数据来自网络（非网络的fileStream除外），为了达到相同的容错能力，通过网络接收到的数据还被复制到其他节点上（默认复制1份，总共2份数据），这就导致错误发生时有两类数据需要恢复，如下。

- **刚收到已经被缓存，但还没有被复制到其他节点的数据**。因为没有其他副本，恢复的唯一方法是从数据源重新获取一份。
- **收到了且已经复制到其他节点的数据**。这部分数据可以从其他节点恢复。

此外，我们还要知道有两类可能发生的错误，如下。

- **worker节点失效**。一旦计算节点失效，所有内存中的数据都会丢失且无法恢复。
- **Driver节点失效**。如果运行Driver进程的节点失效，那么SparkContext也会随之失效，整个Streaming程序会退出，所有附属的执行节点都会退出，内存中的数据全部丢失。

关于容错保障的效果定义，一般都是用数据被计算的次数来定义，分如下3类。

- **至多一次**。每条记录最多被计算一次，或者根本没有计算就丢失了。
- **到少一次**。保证每条记录都不丢失，最少计算一次，但可能会重复多次计算。
- **精准一次**。保证每条记录都不丢失，并且只计算一次，不多不少，显然这是最佳的容错保障。

了解一下通用的流式计算过程有助于了解在每个环节的容错保障效果，一般流式计算分为3步，如图6-9所示。

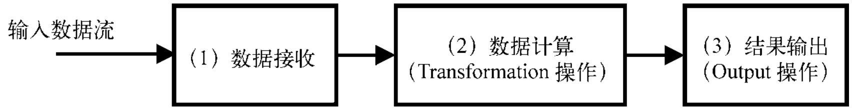


图6-9　流式计算过程

要想实现精准一次的容错效果，我们需要确保每一步都能实现精准一次的计算。

步骤一是数据接收，容错保障很大程序上依赖于数据源，下面将详细讨论。

步骤二是Transformation计算，因为有RDD容错性的保证，所以可以实现精准一次的容错保障。

步骤三是结果输出，默认只提供“至少一次”的容错保障，不能达到“精准一次”的级别，是因为还依赖输出操作的类型和下一级接收系统是否支持事务特性，也在后面单独讨论。

2. 数据接收容错

不同数据源提供不同程序的容错保障。

- 对于HDFS、S3等自带容错功能的文件系统，我们可以保障精准一次的容错能力。
- Spark从1.3版开始引入新的Kafka Direct API，也可以保障精准一次的容错能力。
- 对于其他使用接收器来接收数据的场景，视接收机制是否可靠（有确认机制）以及是否开启WAL功能而不同。如果接收可靠且开启WAL功能，可以确保数据不丢，并且提供实现至少一次级别的容错保障。在开启WAL的情况下，即便数据源接收不可靠，也只会丢失还在缓存的数据并且对其他数据实现至少一次级别的容错保障；其他情况下容错能力都不高，特别是Driver节点失效的情况下基本丧失容错能力，详情参见表6-1。

表6-1　数据源接收可靠性级别对照表

数据源接收可靠性	是否开启WAL	worker节点失效	Driver节点失效
可靠接收	是	数据不丢 至少一次	数据不丢 至少一次
	否	数据不丢 至少一次	已经收到的数据会丢失 无法容错
不可靠接收	是	缓存数据丢失 至少一次	缓存数据丢失 至少一次
	否	缓存数据丢失 至少一次	数据全部丢失 无法容错

3. 结果输出容错

结果输出（`saveAsTextFiles`、`foreachRDD`等）操作本身提供至少一次级别的容错性能，就是说可能输出多次至外部系统，但可能通过一些辅助手段来实现精准一次的容错效果。

当输出为文件时是可以接受的，因为重复的数据会覆盖前面的数据，结果一致，效果相当于精确一次，其他场景下的输出要想实现精确一次的容错，需要一些额外的操作，有如下两种方法。

- **幂等更新**。确保多操作的效果与一次操作的效果相同，比如`saveAs***Files`即便调用多次，结果还是同一个文件。
- **事务更新**。更新时带上事务信息，确保更新只进行一次，比如使用批次时间和RDD的分区编号构造一个事务ID，在更新时使用事务ID来判断是否已经更新，如果已经更新过则跳过，避免重复，实现精准一次的容错效果。

4. 检查点

由于流式计算7×24小时运行的特点，除了考虑具备容错能力，我们还要考虑容错的代价问

题。为了避免错误恢复的代价与运行时间成正比增长，Spark提供了检查点功能，用户定期记录中间状态，避免从头开始计算的漫长恢复。

有一情况下必须启用检查点功能，那就是调用了有状态的Transformation操作，比如updateStateByKey或reduceByKeyAndWindow。因为有状态的操作是从程序开始时一直进行的，如果不做检查点，那么计算链接会随着时间一直增长，重新计算的代价也将会是天文数字。

另外，如果期望程序在因Driver节点失效后的重启之后可以继续运行，也建议开启检查点功能，可以记录配置、操作以及未完成的批次，这样重启后可以继续运行。

当然，不启用检查点功能也是可以的，实际上大部分程序都不需要，这可视需求来选择。

开启检查点的方法很简单，调用streamingContext.checkpoint(checkpointDirectory)即可，参数是一个支持容错的文件系统目录，可以是HDFS或S3。

开启之后，前面提到的有状态的Transformation操作就可以调用了。

为了让Driver程序自动重启时也能使用检查点功能，还需要添加一些代码。参考下面的示例，主要改动是在启动后创建StreamingContext时检查一下检查点目录，如果存在则从这个目录恢复StreamingContext，否则就创建新的。示例代码：

```
// 通过函数来创建StreamingContext
def functionToCreateContext(): StreamingContext = {
    val ssc = new StreamingContext(...)   // 新的context
    ssc.checkpoint(checkpointDirectory)   // 设置检查点目录

    val lines = ssc.socketTextStream(...) // 创建DStream
    // 其他处理逻辑
    ...
    ssc // 返回context
}

// 从检查点目录恢复StreamingContext或创建一个新的
val ssc = StreamingContext.getOrCreate(checkpointDirectory,
    functionToCreateContext _)
ssc.start()
ssc.awaitTermination()
```

注意，代码执行的前提是在程序发布时配置了自动重启，参考前面的介绍。

注意，检查点是有代价的，需要存储数据至存储系统，增加批次的计算时间，并且降低吞吐量，我们可以通过运行时的监控来查看实际的额外负载。我们还可以通过增加周期的时间间隔来降低影响，一般建议时间间隔至少为10秒。

6.2.3　性能调优

Spark流式计算程序要想运行顺畅，也需要一些基本的调优，总结一下主要在两个方向上：

- 每个批次的处理时间尽可能短；
- 收到数据后，尽可能快地处理。

1. 减少批处理的时间

有很多方法可以用于优先计算，减少处理的时间，这里重点讨论流式计算这一特定场景下最重要的几个方法。

一是增加数据接收的并发数量，尤其是当瓶颈发生在数据接收的时候。默认每个Input `DStream`都只会创建一个接收器，运行在某个节点上，我们可以创建多个Input `DStream`，让它们接收不同的数据分区，以实现并行接收。比如一个接收两个topic的Kafka Input `DStream`可以优化成两个Input `DStream`，各接收一个topic，然后再合并，示例如下：

```
val numStreams = 5
val kafkaStreams = (1 to numStreams).map { i => KafkaUtils.createStream(...) }
val unifiedStream = streamingContext.union(kafkaStreams)
unifiedStream.print()
```

二是数据处理的并发度，如果并发度不够，可能导致集群的资源不被充分利用。一个最简单的方法是看各机器CPU的所有核心是不是都在工作，如果有空闲的，则可以考虑增加并发度（可以调整选项`spark.default.parallelism`）。

三要数据序列化，数据收到后，当需要与磁盘交换数据时，数据可能会进行序列化和反序列化，好处是节省空间和内存，但会增加计算负载。因此，我们应尽可能地使用Kryo来完成这项工作，CPU和内存开销都相对少一些。

最后是要注意task启动的额外开销，如果task启动过于频繁（比如每秒50次），那么额外的开销可能非常高，甚至无法达到那样的实时计算要求。Standalone模式和Mesoscoarse-grained模式下开销相对会小一些。

2. 设置合理批次间隔时间

为了让每个批次的数据能够尽快处理，批次间隔时间的设置非常重要。经验表明，一般来说短时间间隔会导致更多的额外开销，以及无法完成的风险，所以前期可以采取相对保守的方法，比如将间隔设置为5~10秒。然后，我们通过观察运行数据或者最终的输出数据确保系统足够实时，每个间隔的实际计算时间远小于间隔时间，然后再逐渐按需要缩短间隔时间。

6.2.4 与 Storm 的对比

Spark一直是后来者，在批处理领域出现在Hadoop MR之后，但凭借优秀的性能表现大有超越之势。它在流式计算领域也是后来者，在Spark Streaming出现之前，业界已经有了Storm并且应用广泛，Storm在2014年9月从0.9.1版本开始正式成为Apache旗下的顶级项目。

“他山之石，可以攻玉。”这里对两者做一下简单对比，也让读者对Spark Streaming有更深入

的认识。

Storm是开源免费的分布式实时计算系统，可以简单、实时、可靠地处理海量的数据流，具有高性能、可扩展、自动容错、确保数据不丢失等特性。它支持非常多的语言，可以应用在实时分析、在线实时机器学习、分布式RPC、ETL等众多领域，其结构如图6-10所示。

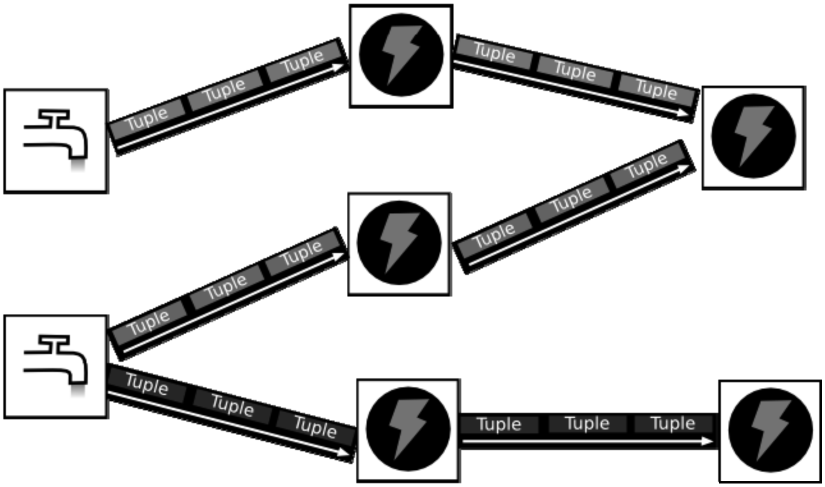


图6-10　Storm结构示意图（另见彩插图6-10）

大体上两者非常接近，而且都处于快速迭代过程中，即便一时的对比可能某一方占优势，但后者可能很快就追赶上来。比如在性能方面，Spark Streaming刚发布不久，有基准测试显示性能超过Storm几十倍，原因是Spark Streaming采用了小批量模式，而Storm是一条消息一条消息地计算。但后来Storm也推出了称为Trident的小批量计算模式，性能应该不是差距了。而且双方都在持续更新，底层的一个通信框架的更新或者某个路径的代码优化都可能让性能有较大的提升。

但是两者的基因不同，更具体地说就是核心数据抽象不同。这是无法改变的，而且也不会轻易改变，这样的基因也决定了它们各自最适合的应用场景。前面已经讲过，Spark Streaming的核心抽象是`DSTream`，里面是RDD，下层是Spark核心DAG调度，所以Spark Streaming的这一基因决定了其粒度是小批量的，无法做更精细地控制。比如，调度周期1秒可能已经达到极限了，更小的周期意义已经不大。数据的可靠性也是以批次为粒度的，但好处也很明显，就是有可能实现更大的吞吐量。另外，得益于Spark平台的良好整合性，完成相同任务的流式计算程序与历史批量处理程序的代码基本相同，而且还可以使用平台上的其他模块比如SQL、机器学习、图计算的计算能力，在开发效率上占有优势。而Storm的核心数据抽象是tuple，是命名的值列表，相当于消息，不是分布式的，所以Storm更擅长细粒度的消息级别的控制，比如延时可以实现毫秒级，数据可靠性也是以消息为粒度的。

核心数据抽象的不同导致了它们在计算模式上的本质区别。Spark Streaming在本质上其实是

像MR一样的批处理计算，但将批处理的周期从常规的几十分钟级别尽可能缩短至秒级，也算达到了实时计算的延时指标。而且，它支持各类数据源，基本可以实现流式计算的功能，但延时无法进一步缩短了。但Storm的设计初衷就是实时计算，毫秒级的计算当然不在话下，而且后期通过更高级别的Trident也实现了小批次处理功能。

表6-2是两者的详细对比，仅供参考。

表6-2　Spark Streaming与Storm对比

	Spark Streaming	Storm
计算模型	小批量计算	实时数据流计算
实时性	秒级	最快10毫秒级
数据可靠性	精确只被处理一次	核心支持最少一次或最多一次； Trident支持全部最少一次、最多一次、精确一次
性能	性能测试视硬件、网络、程序等多方因素而定，无法直接对比，这里只是引用一些公开的数据： Spark Streaming：40万记录/秒/节点 Storm：100万/秒/节点	
实现语言	Scala	Clojure
编程语言	Scala、Java、Python	Trident只支持Clojure、Java、Scala，核心还支持Python、Ruby等其他语言
相同点	分布式、容错、可扩展	

6.3　应用场景：一个类似百度统计的流式实时系统

我们知道网站用户访问流量是不间断的，基于网站的访问日志，即Web log分析是典型的流式实时计算应用场景。比如百度统计，它可以做流量分析、来源分析、网站分析、转化分析。另外还有特定场景分析，比如安全分析，用来识别CC攻击、SQL注入分析、脱库等。这里我们简单实现一个类似于百度分析的系统。

6.3.1　Web log 实时统计场景

百度统计（baidu.tongji.com）是百度推出的一款免费的专业网站流量分析工具，能够告诉用户访客是如何找到并浏览用户的网站的，以及在网站上浏览了哪些页面。这些信息可以帮助用户改善访客在其网站上的使用体验，不断提升网站的投资回报率。

百度统计提供了几十种图形化报告，包括：趋势分析、来源分析、页面分析、访客分析、定制分析等多种统计分析服务。

这里我们参考百度统计的功能，基于Spark Streaming简单实现一个分析系统，使之包括以下分析功能。

- **流量分析**。一段时间内用户网站的流量变化趋势，针对不同的IP对用户网站的流量进行细分。常见指标是总PV和各IP的PV。
- **来源分析**。各种搜索引擎来源给用户网站带来的流量情况，需要精确到具体搜索引擎、具体关键词。通过来源分析，用户可以及时了解哪种类型的来源为其带来了更多访客。常见指标是搜索引擎、关键词和终端类型的PV。
- **网站分析**。各个页面的访问情况，包括及时了解哪些页面最吸引访客以及哪些页面最容易导致访客流失，从而帮助用户更有针对性地改善网站质量。常见指标是各页面的PV。

6.3.2　日志实时采集

Web log一般在HTTP服务器收集，比如Nginx access日志文件。一个典型的方案是Nginx日志文件 + Flume + Kafka + Spark Streaming，如下所述：

(1) 接收服务器用Nginx，根据负载可以部署多台，数据落地至本地日志文件；
(2) 每个Nginx节点上部署Flume，使用`tail -f`实时读取Nginx日志，发送至KafKa集群；
(3) 专用的Kafka集群用户连接实时日志与Spark集群，详细配置可以参考http://spark.apache.org/docs/1.4.1/streaming-kafka-integration.html；
(4) Spark Streaming程序实时消费Kafka集群上的数据，实时分析，输出；
(5) 结果写入MySQL数据库。

当然，还可以进一步优化，比如CGI程序直接发日志消息到Kafka，节省了写访问日志的磁盘开销。这里主要专注Spark Streaming的应用，所以我们不做详细论述。

6.3.3　流式分析系统实现

我们简单模拟一下数据收集和发送的环节，用一个Python脚本随机生成Nginx访问日志，并通过脚本的方式自动上传至HDFS，然后移动至指定目录。Spark Streaming程序监控HDFS目录，自动处理新的文件。

生成Nginx日志的Python代码如下，保存为文件sample_web_log.py。

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import random
import time

class WebLogGeneration(object):

    def __init__(self):
        # 前面7条是IE，所以大概浏览器类型70%为IE。接入类型上，20%为移动设备，分别是7条和8条，5%
        # 为空
        self.user_agent_dist = {
```

```
            0.0:"Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; Trident/6.0)",
            0.1:"Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; Trident/6.0)",
            0.2:"Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0; .NET
                CLR 2.0.50727)",
            0.3:"Mozilla/4.0 (compatible; MSIE6.0; Windows NT 5.0; .NET CLR
                1.1.4322)",
            0.4:"Mozilla/5.0 (Windows NT 6.1; Trident/7.0; rv:11.0) like Gecko",
            0.5:"Mozilla/5.0 (Windows NT 6.1; WOW64; rv:41.0) Gecko/20100101
                Firefox/41.0",
            0.6:"Mozilla/4.0 (compatible; MSIE6.0; Windows NT 5.0; .NET CLR
                1.1.4322)",
            0.7:"Mozilla/5.0 (iPhone; CPU iPhone OS 7_0_3 like Mac OS X)
                AppleWebKit/537.51.1 (KHTML, like Gecko) Version/7.0 Mobile/11B511
                Safari/9537.53",
            0.8:"Mozilla/5.0 (Linux; Android 4.2.1; Galaxy Nexus Build/JOP40D)
                AppleWebKit/535.19 (KHTML, like Gecko) Chrome/18.0.1025.166 Mobile
                Safari/535.19",
            0.9:"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_5) AppleWebKit/537.36
                (KHTML, like Gecko) Chrome/45.0.2454.85 Safari/537.36",
            1:" ",
        }
        self.ip_slice_list = [10, 29, 30, 46, 55, 63, 72, 87,
           98,132,156,124,167,143,187,168,190,201,202,214,215,222]
        self.url_path_list =
           ["login.php","view.php","list.php","upload.php","admin/login.php","edit.
           php","index.html"]
        self.http_refer = [
            "http://www.baidu.com/s?wd={query}",
            "http://www.google.cn/search?q={query}",
            "http://www.sogou.com/web?query={query}",
            "http://www.yahoo.com/s?p={query}",
            "http://cn.bing.com/search?q={query}"
        ]
        self.search_keyword = ["spark","hadoop","hive","spark mlib","spark sql"]

    def sample_ip(self):
        slice = random.sample(self.ip_slice_list, 4) # 从ip_slice_list中随机获取4个
                                                     # 元素，作为一个片断返回
        return  ".".join([str(item) for item in slice])

    def sample_url(self):
        return  random.sample(self.url_path_list,1)[0]

    def sample_user_agent(self):
        dist_uppon = random.uniform(0, 1)
        return self.user_agent_dist[float('%0.1f' % dist_uppon)]

    # 主要搜索引擎referrer参数
    def sample_refer(self):
        if random.uniform(0, 1) > 0.2:  # 只有20%流量有refer
            return "-"

        refer_str=random.sample(self.http_refer,1)
        query_str=random.sample(self.search_keyword,1)
```

```
        return refer_str[0].format(query=query_str[0])

    def sample_one_log(self,count = 3):
        time_str = time.strftime("%Y-%m-%d %H:%M:%S",time.localtime())
        while count >1:
            query_log = "{ip} - - [{local_time}] \"GET /{url} HTTP/1.1\" 200 0
                \"{refer}\" \"{user_agent}\" \"-\"".format(ip=self.sample_ip(),
            local_time=time_str,url=self.sample_url(),refer=
            self.sample_refer(),user_agent=self.sample_user_agent())
            print query_log
            count = count -1

if __name__ == "__main__":
    web_log_gene = WebLogGeneration()

    web_log_gene.sample_one_log(random.uniform(30000, 50000))
```

这是一条日志的示例，为一行形式，各字段间用空格分隔，字符串类型的值用双引号包围：

```
46.202.124.63 - - [2015-11-26 09:54:27] "GET /view.php HTTP/1.1" 200 0
"http://www.google.cn/search?q=hadoop" "Mozilla/5.0 (compatible; MSIE 10.0; Windows
NT 6.2; Trident/6.0)" "-"
```

然后需要一个简单的脚本来调用上面的脚本以随机生成日志，上传至HDFS，然后移动到目标目录：

```
#!/bin/bash

# HDFS命令
HDFS="/usr/local/myhadoop/hadoop-2.6.0/bin/hadoop fs"

# Streaming程序监听的目录，注意跟后面Streaming程序的配置要保持一致
streaming_dir="/spark/streaming"

# 清空旧数据
$HDFS -rm "${streaming_dir}"'/tmp/*' > /dev/null 2>&1
$HDFS -rm "${streaming_dir}"'/*'     > /dev/null 2>&1

# 一直运行
while [ 1 ]; do
    ./sample_web_log.py > test.log

    # 给日志文件加上时间戳，避免重名
    tmplog="access.`date +'%s'`.log"

    # 先放在临时目录，再move至Streaming程序监控的目录下，确保原子性
    # 临时目录用的是监控目录的子目录，因为子目录不会被监控
    $HDFS -put test.log ${streaming_dir}/tmp/$tmplog
    $HDFS -mv           ${streaming_dir}/tmp/$tmplog ${streaming_dir}/

    echo "`date +"%F %T"` put $tmplog to HDFS succeed"
    sleep 1
done
```

Spark Streaming程序代码如下所示，可以在bin/spark-shell交互式环境下运行，如果要以Spark程序的方式运行，按注释中的说明调整一下`StreamingContext`的生成方式即可。启动bin/spark-shell时，为了避免因`DEBUG`日志信息太多而影响观察输出，可以将`DEBUG`日志重定向至文件，屏幕上只显示主要输出，方法是`./bin/spark-shell 2>spark-shell-debug.log`：

```
import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds, StreamingContext}

// 设计计算的周期，单位：秒
val batch = 10

/*
 * 这是bin/spark-shell交互式模式下创建StreamingContext的方法
 * 非交互式请使用下面的方法来创建
 */
val ssc = new StreamingContext(sc, Seconds(batch))

/*
// 非交互式下创建StreamingContext的方法
val conf = new SparkConf().setAppName("NginxAnay")
val ssc = new StreamingContext(conf, Seconds(batch))
*/



/*
 * 创建输入DStream，是文本文件目录类型
 * 本地模式下也可以使用本地文件系统的目录，比如file:///home/spark/streaming
 */
val lines = ssc.textFileStream("hdfs:///spark/streaming")

/*
 * 下面是统计各项指标，调试时可以只进行部分统计，方便观察结果
 */


// 1. 总PV
lines.count().print()


// 2. 各IP的PV，按PV倒序
//    空格分隔的第一个字段就是IP
lines.map(line => {(line.split(" ")(0), 1)}).reduceByKey(_ + _).transform(rdd => {
    rdd.map(ip_pv => (ip_pv._2, ip_pv._1)).
    sortByKey(false).
    map(ip_pv => (ip_pv._2, ip_pv._1))
}).print()


// 3. 搜索引擎PV
val refer = lines.map(_.split("\"")(3))
```

```
// 先输出搜索引擎和查询关键词，避免统计搜索关键词时重复计算
// 输出(host, query_keys)
val searchEnginInfo = refer.map(r => {

    val f = r.split('/')

    val searchEngines = Map(
        "www.google.cn" -> "q",
        "www.yahoo.com" -> "p",
        "cn.bing.com" -> "q",
        "www.baidu.com" -> "wd",
        "www.sogou.com" -> "query"
    )

    if (f.length > 2) {
        val host = f(2)

        if (searchEngines.contains(host)) {
            val query = r.split('?')(1)
            if (query.length > 0) {
                val arr_search_q =
                    query.split('&').filter(_.indexOf(searchEngines(host)+"=") == 0)
                if (arr_search_q.length > 0)
                    (host, arr_search_q(0).split('=')(1))
                else
                    (host, "")
            } else {
                (host, "")
            }
        } else
            ("", "")
    } else
        ("", "")
})

// 输出搜索引擎PV
searchEnginInfo.filter(_._1.length > 0).map(p => {(p._1, 1)}).reduceByKey(_ +
    _).print()


// 4. 关键词PV
searchEnginInfo.filter(_._2.length > 0).map(p => {(p._2, 1)}).reduceByKey(_ +
    _).print()


// 5. 终端类型PV
lines.map(_.split("\"")(5)).map(agent => {
    val types = Seq("iPhone", "Android")
    var r = "Default"
    for (t <- types) {
        if (agent.indexOf(t) != -1)
            r = t
    }
    (r, 1)
```

```
}).reduceByKey(_ + _).print()


// 6. 各页面PV
lines.map(line => {(line.split("\"")(1).split(" ")(1), 1)}).reduceByKey(_ +
    _).print()



// 启动计算,等待执行结束（出错或Ctrl-C退出）
ssc.start()
ssc.awaitTermination()
```

打开两个终端，一个调用上面的bash脚本模拟提交日志，一个在交互式环境下运行上面的Streaming程序。你可以看到各项指标的输出，比如某个批次下的输出为（依次对应上面的6个计算项）：

```
1. 总PV
-------------------------------------------
Time: 1448533850000 ms
-------------------------------------------
44374

2. 各IP的PV，按PV倒序
-------------------------------------------
Time: 1448533850000 ms
-------------------------------------------
(72.63.87.30,30)
(63.72.46.55,30)
(98.30.63.10,29)
(72.55.63.46,29)
(63.29.10.30,29)
(29.30.63.46,29)
(55.10.98.87,27)
(46.29.98.30,27)
(72.46.63.30,27)
(87.29.55.10,26)
...

3. 搜索引擎PV
-------------------------------------------
Time: 1448533850000 ms
-------------------------------------------
(cn.bing.com,1745)
(www.baidu.com,1773)
(www.google.cn,1793)
(www.sogou.com,1845)

4. 关键词PV
-------------------------------------------
Time: 1448533850000 ms
-------------------------------------------
(spark,1426)
```

```
(hadoop,1455)
(spark sql,1429)
(spark mlib,1426)
(hive,1420)
```

5. 终端类型PV

```
-------------------------------------------
Time: 1448533850000 ms
-------------------------------------------
(Android,4281)
(Default,35745)
(iPhone,4348)
```

6. 各页面PV

```
-------------------------------------------
Time: 1448533850000 ms
-------------------------------------------
(/edit.php,6435)
(/admin/login.php,6271)
(/login.php,6320)
(/upload.php,6278)
(/list.php,6411)
(/index.html,6309)
(/view.php,6350)
```

查看数据更直观的做法是用图形来展示，常见做法是将结果写入外部DB，然后通过一些图形化报表展示系统展示出来。比如对于终端类型，我们可以用饼图展示，如图6-11所示。

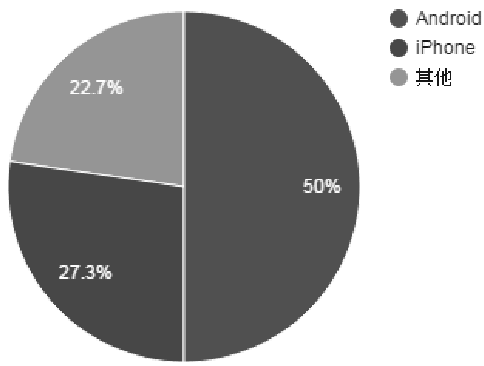


图6-11　终端类型分布图示例（另见彩插图6-11）

对于连续的数据，我们也可以用拆线图来展示趋势。比如某页面的PV，如图6-12所示。

除了常规的每个固定周期进行一次统计，我们还可以对连续多个周期的数据进行统计。以统计总PV为例，上面的示例是每10秒统计一次，可能还需要每分钟统计一次，相当于6个10秒的周期。我们可以利用窗口方法实现，不同的代码如下：

```
// 窗口方法必须配置checkpint，可以这样配置：
ssc.checkpoint("hdfs:///spark/checkpoint")

// 这是常规每10秒一个周期的PV统计
lines.count().print()

// 这是每分钟（连续多个周期）一次的PV统计
lines.countByWindow(Seconds(batch*6), Seconds(batch*6)).print()
```

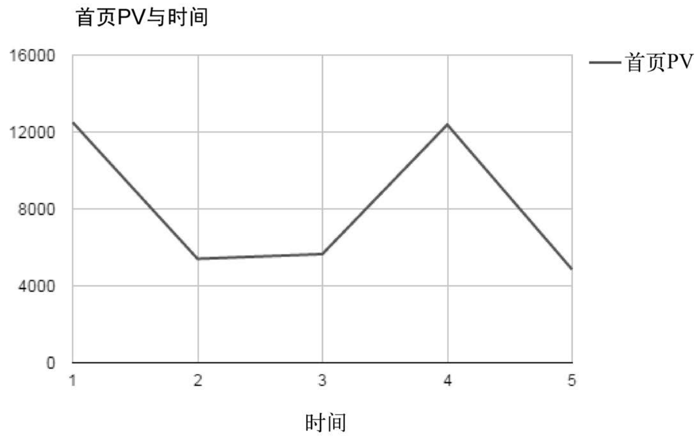


图6-12　首页PV趋势图示例（另见彩插图6-12）

使用相同的办法运行程序之后，我们首先会看到连续6次10秒周期的PV统计输出：

```
-------------------------------------------
Time: 1448535090000 ms
-------------------------------------------
1101

-------------------------------------------
Time: 1448535100000 ms
-------------------------------------------
816

-------------------------------------------
Time: 1448535110000 ms
-------------------------------------------
892

-------------------------------------------
Time: 1448535120000 ms
-------------------------------------------
708

-------------------------------------------
Time: 1448535130000 ms
-------------------------------------------
881
```

```
-------------------------------------------
Time: 1448535140000 ms
-------------------------------------------
872
```

在这之后，有一个1分钟周期的PV统计输出，它的值刚好是上面6次计算结果的总和：

```
-------------------------------------------
Time: 1448535140000 ms
-------------------------------------------
5270
```

第 7 章

Spark图计算

7

作为一个通用的计算框架，Spark有很多应用场景，我们选一个比较有意思的来做一次实战演练。“大数据”概念越来越火爆，随之SNA（Social Network Analysis，社交网络分析）也成为了一项热门研究。Spark在这方面做了充分的准备，自身集成了一套成熟的计算库GraphX，来解决类似的计算问题。下面，我们就使用Spark + GraphX（其标志参见图7-1）做一次社交网络分析。

图7-1　GraphX的标志（来源于Spark官网）

7.1　什么是图计算

图计算，可以简单理解为以图这种数据结构为基础，基于其实现的相关算法及应用。社交网络中人与人之间的关系，如果用计算机数据结构表示，最合适的就是图了。其中，图的“顶点”表示社交中的人，“边”则表示人和人的关系。所以要做社交网络分析，我们先要了解图计算，这是整个分析的基础。也正因此，Spark的图计算库叫作GraphX。

7.1.1　图的基本概念

图是基础的数据结构。和链表、树不同，它是一种非线性数据结构。其基本结构很简单，如图7-2所示。

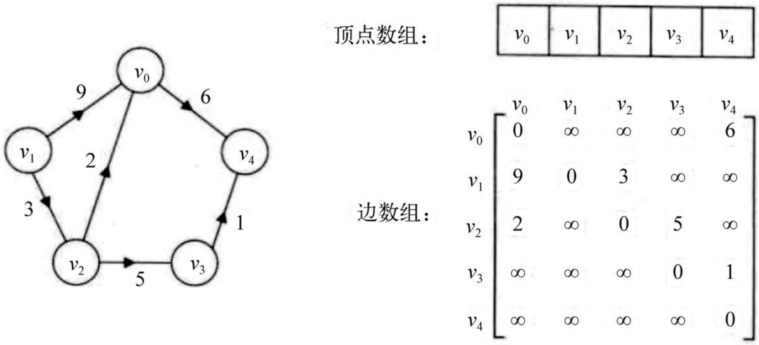


图7-2　图的表示方法

一个图由顶点集V和顶点间的关系集合E（边的集合）组成，可以用二元组定义为：G=(V,E)。

图中各数据元素之间的关系可以是任意的，且它描述的是“多对多”的关系。

例如，图7-2中顶点的集合就表示为：

$$V(G)=\{v_0, v_1, v_2, v_3, v_4\}$$

边的集合则表示为：

$$E(G)=\{(v_0, v_4), (v_3, v_4), (v_2, v_3), (v_2, v_0),(v_1, v_2), (v_1, v_0)\}$$

这是一个有向图。边有方向，(v_1, v_2)和(v_2, v_1)表示不同的两条边。若边无方向，则是无向图。

基于图的数据结构衍生出了很多基础算法，如遍历、最小生成树、最短路径等。比如，教科书中著名的Prim算法和Kruskal算法，就是计算最小生成树。

业界也有很多开源的图计算库，常见的如Python的NetworkX、Spark的GraphX等。这些库基本都提供3类API接口，如下。

- **图生成**。将文本、日志等数据转换为图的数据结构。
- **访问图数据**。查询定点数、边数；计算某个定点的出度和入度等。
- **图算法**。遍历图节点和边、计算图的连通性、计算最大子图、图的合并等基本算法。

7.1.2　图计算的应用

基于图的结构有很多应用场景，比如淘宝的推荐商品、腾讯的推荐好友，再比如一些网络路由算法、SNA、Language Modeling等也都会用到图计算。

图的计算量一般都比较大，而且通常会有多次迭代。比如，若简单地计算顶点出入度，时间复杂度就是$O(n \times m)$。像BAT这种公司上亿的数据量级，基本是不可完成的任务。如果能将算法并行化，利用机器数量弥补速度，这将会是件美好的事情。

7.2 Spark GraphX 简介

为了提高图计算的速度，很多企业、社区都提供了并行的图计算解决方案，常见的有Pregel、Power Graph、Graphlib等。当然，Spark的GraphX是其中的新秀和佼佼者。它依托Spark的强大计算能力，提供了图计算需要的便捷API，同时兼具并行计算的性能，是做大规模图计算的一把利器。

7.2.1 GraphX 实现

众所周知，Spark抽象了一个通用的数据结构RDD（Resilient Distributed Dataset，弹性分布式数据集）来代表运算中需要的各种数据类型。GraphX的核心数据结构则是Graph，这是一种携带了每个点和边的属性的有向多重图（directed multigraph）。所谓"多重图"，就是一对源、目的节点之间允许存在多条边，以便表示不同的关系（如既是同学，又是同事）。下面是Graph的定义：

```
class Graph[VD, ED] {
    val vertices: VertexRDD[VD]
    val edges: EdgeRDD[ED]
}
```

`Graph`的数据结构比较简单，由`VertexRDD[VD]`和`EdgeRDD[ED]`组成，`VD`和`ED`分别表示顶点和边的抽象数据结构，实际等价于`RDD[(VertexID, VD)]`和`RDD[Edge[ED]]`这两种RDD（Resilient Distributed Dataset，弹性分布式数据集；RDD是Scala语言的核心数据结构，详情可参考Scala相关图书）。

GraphX内部是如何存储一个图的？它借鉴了PowerGraph，使用了Vertex-Cut（点分割）方式，即将图的顶点集合划分到不同的计算节点上，这样可以减少分布式计算时的通信和存储消耗。

一个图的点分割存储方式，如图7-3所示。

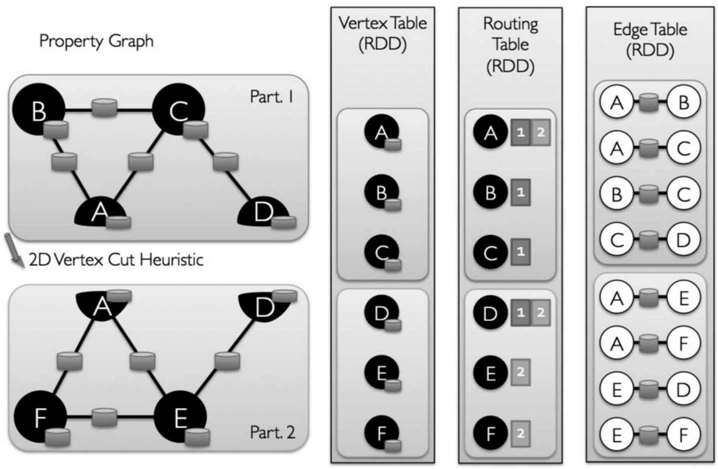


图7-3 图的点分割存储方式（来源于Spark官网）

GraphX内部维护了3个RDD来存储一个图：一个是顶点表（Vertex Table），存储顶点信息；一个是边表（Edge Table），存储边信息；一个是路由表（Routing Table），用来查询顶点存储在哪个计算节点上。按Vertex-Cut方式，顶点信息仅会存一份，*A*、*B*、*C*顶点存储在节点1（Part 1）上，*D*、*E*、*F*存储在节点2（Part 2）上。当需要把边和顶点做关联计算时，比如计算出入度，则按如下方式计算：

(1) 遍历Edge Table，获取每条边的顶点ID；

(2) 根据顶点ID查询Routing Table，找到该顶点的存储节点；

(3) 将边信息广播到顶点所属的存储节点上，做并行计算；

(4) 输出每个顶点的计算结果。

7.2.2　GraphX 常用 API 介绍

GraphX的API分为几类：数据查询、数据转换、结构转换、关联聚合、缓存操作等。基本上图的操作，只需要一个函数调用即可完成，大大精简了图计算的程序设计。这样一来，我们可以聚焦在算法逻辑实现上，而不用太关心数据结构本身。

在图计算中最常用的API有3类：数据查询类、关联类和聚合类。下面我们简单介绍下这3类操作，分别如表7-1、表7-2和表7-3所示。

表7-1　数据查询类操作

数据查询类操作	说　　明
`val numEdges: Long`	获取边数
`val numVertices: Long`	获取节点数
`val inDegrees: VertexRDD[Int]`	获取入度
`val outDegrees: VertexRDD[Int]`	获取出度
`val degrees: VertexRDD[Int]`	获取总度数

数据查询类API主要用来获取图的基础信息，比较简单。

表7-2　关联类操作

关联类操作	说　　明
`def joinVertices[U](table: RDD[(VertexID, U)])(mapFunc: (VertexID, VD, U) => VD): Graph[VD, ED]`	将一个RDD和图的节点集合进行`join`运算。即在RDD包含的节点上，调用map函数分布式计算，返回一个新图
`def outerJoinVertices[U, VD2](other: RDD[(VertexID, U)]) (mapFunc: (VertexID, VD, Option[U]) => VD2) :Graph[VD2, ED]`	这个函数会更通用。和`joinVertices`运算方式类似，但`map`函数会在所有的vertex节点上执行。我们可以用这个做一些图节点的初始化操作

关联类API是将一个图和一个RDD通过节点ID（`VertexID`）关联，来使图获得RDD中的信息。

表7-3 聚合类操作

聚合类操作	说 明

`def aggregateMessages[Msg: ClassTag](`
`sendMsg: EdgeContext[VD, ED, Msg] => Unit,`
`mergeMsg: (Msg, Msg) =>Msg,`
`tripletFields:`
`TripletFields = TripletFields.All)`
`: VertexRDD[A]`
	分布式遍历每条边，执行自定义的sendMsg计算；然后在边所属的节点上，执行自定义的mergeMsg计算

聚合类API的作用不太容易理解。它的一个典型应用场景就是计算每个节点的出度和入度：sendMsg会把边EdgeContext的权值信息Msg分别发送到该有向边的源节点和目的节点，通过mergeMsg累加即可得到各节点的出度入度。来看以下代码：

```
    // 计算每个目的节点的入度，作为权值（原算法是把出入度都相加）
    // (Long, Long)表示这个节点的（入度，出度）
    val nodeWeightMapFuncEx = (ctx:EdgeContext[VD, Long, (Long,Long)]) => {
      ctx.sendToDst((ctx.attr,0L))
      ctx.sendToSrc((0L,ctx.attr))
    }
    // 对一个节点的边信息做聚合。入度相加，出度相加
    val nodeWeightReduceFuncEx = (e1:(Long, Long),e2:(Long,Long)) => (e1._1+e2._1,
        e1._2+e2._2)

    val nodeWeights = graph.aggregateMessages(nodeWeightMapFuncEx,
nodeWeightReduceFuncEx)
```

这样nodeWeights: VertexRDD[(Long, Long)]就包含了每个节点的入度、出度值。是不是很方便？

7.3 应用场景：基于新浪微博数据的社交网络分析

了解了图计算和GraphX，下面我们正式进入社交网络分析的实战。这里的例子使用大家很熟悉的新浪微博数据，来阐释社交网络分析的原理，并使用GraphX实现分析算法。

7.3.1 社交网络分析的主要应用

在分析复杂的社会、技术以及信息系统时，我们常把这些系统描述成网络的形式。在这样的关系网络中，节点代表一个成员，而边就代表成员之间的关系。在现实生活中，一个人可能从属于不同的团体，比如其工作的部门、生活的小区、同学圈子、亲戚圈子、同事圈子等。如果能把这些不同的圈子自动分析出来，大到舆情监控，小到推荐好友以扩展新用户、精准广告投放等，这些场景都能通过社交网络分析来提升效果。结合目前日益火爆的“大数据”，这就是应用大数据来解决实际问题的一个很好的例子。

所谓“圈子”，就是把人归属到不同的群体中。落实到算法上，划分圈子就是根据网络的属性特征把关系网络中各个的节点划分到各个具有不同含义的集合中。不同的圈子内部节点之间的连接很紧密，而圈子之间的连接相对比较松散。

通过将社会生活中的关系网络抽象成计算机能够表示的图，之后运用社区发现算法， 根据图的某种特性将图划分成一个个子图（即社区），这就达到了自动识别不同圈子的目的。

7.3.2　社区发现算法简介

社区发现的算法多种思路，比较常见的有两种：一种是分离的思路，就是找出社区之间的边，把这些边从图中移除；一种是聚合思路，将联系紧密的节点聚合为一个社区，并通过优化某个相关变量的函数来实现聚合。

前人已在这两个思路上有了大量的研究，而根据这两类算法的结果看，聚合的思路比分离思路好，且算法的效率也较高。因此，聚合算法吸引很多学者做了大量相关研究，逐步形成了现在的社区发现算法。比如，密歇根大学的M. E. J. Newman和康奈尔大学的M. Girvan，他们在2003年提出了一个基于模块属性的测量方法。他们在算法中引入了一个变量（该变量称作模块度），用于衡量社区划分结果的合理性。其原理是用某种划分结果的模块内聚性与随机划分结果的内聚性的差值，对划分结果进行评估，找到模块内聚性最优的划分。虽然寻找最优解往往非常困难，但这个思路给大家指引了优化方向。模块度的思路对后来的社区发现算法有很重要的影响，很多有影响的算法都是基于该特性进行算法设计的。

2008年，以比利时鲁汶大学的Vincent D. Blondel为主的几位学者，提出了基于模块度的一个快速算法——Louvain算法（也就是Blondel所在大学的名字）。该算法可以快速处理具有数以亿计节点的网络，用模块度对社区划分的质量进行衡量，其值位于–1到1之间。模块度的值越大，说明社区划分的质量越高。

$$Q=\frac{1}{2m}\sum_{i,j}[A_{ij}-\frac{K_iK_j}{2m}]\delta(c_i,c_j)$$

$$\delta(u,v)=\begin{cases}1,\ \text{当}\,u==v\\0,\ \text{其他情况}\end{cases}$$

其中，A_{ij}是节点i和节点j之前边的权重，如果图里的边都没有权重，就可以看作1。$k_i=\sum_j A_{ij}$ 表示所有与节点i相连的边权重之和，即节点i的度数。c_i表示节点i所属的社区；$m=\frac{1}{2}\sum_{ij}A_{ij}$ 表示所有边的权重之和，也就是边的数目。

公式中的核心部分：$A_{ij}-\frac{K_iK_j}{2m}$ 应该如何理解呢？ $\frac{K_j}{2m}$ 代表任一节点与节点j相连的概率。现

在节点i的度数是K_i，所以随机情况下，节点i与节点j相连的度数就是$\frac{K_iK_j}{2m}$。$A_{ij}-\frac{K_iK_j}{2m}$值越大，代表当前社区划分与随机情况下差距越大，划分的效果就越好。这里的Q，就是模块度了。

如果用GraphX实现Louvain算法，重点就在算法实现上，数据结构非常简单。

Louvain算法中的主要数据结构VertexState代码如下：

```
class VertexState extends Serializable{

    var community = -1L          // 所属社区ID
    var communitySigmaTot = 0L // 社区的度数
    var internalWeight = 0L     // 节点总度数
    var nodeWeight = 0L;        // 节点的出度
    var changed = false

    override def toString(): String = {
        "{community:"+community+",communitySigmaTot:"+communitySigmaTot+
        ",internalWeight:"+internalWeight+",nodeWeight:"+nodeWeight+"}"
    }
}
```

程序的流程也很简单，如下。

(1) 将图中的每个节点看成一个独立的社区，此时社区的数目与节点个数相同；

(2) 对每个节点i，依次尝试将其分配到每个邻居节点所在的社区，计算分配前与分配后的模块度变化$\Delta\delta$，并记录$\Delta\delta$最大的那个邻居节点，如果$\max\Delta\delta>0$，则把节点i分配给$\Delta\delta$最大的那个邻居节点所在的社区，否则保持不变；

(3) 重复步骤(2)，直到所有节点的所属社区不再变化；

(4) 对图进行压缩，将所有在同一个社区的节点压缩成一个新节点，社区内节点之间的边的权重转化为新节点的环的权重，社区间的边权重转化为新节点间的边权重（这一步主要为减少后续计算量）；

(5) 重复步骤(1)，直到整个图的模块度不再发生变化。

每轮迭代都会把节点按社区压缩，这就减少了边和节点的数目。计算节点i到其邻居节点j模块度的变化，只与节点i、j的社区有关，与其他社区无关，这样一来就加快了计算的速度。那么如何判断一个节点i是否属于社区c_i呢？

$$\Delta\delta=\left[\frac{\sum_{in}+K_{i,in}}{2m}-\left(\frac{\sum_{tot}-K_i}{2m}\right)^2\right]-\left[\frac{\sum_{in}}{2m}-\left(\frac{\sum_{tot}}{2m}\right)^2-\left(\frac{K_i}{2m}\right)^2\right]$$

其中$\sum_{in}$表示社区c内的边权重之和，$\sum_{tot}$表示所有与社区c内的节点相连的边权重之和。$\Delta\delta$的计算分两部分，前一个中括号代表把节点i加入社区c的模块度，后一个中括号表示节点i作为一个独立社区的模块度，$\Delta\delta$最大的那个社区就是节点i所属的新社区。

大家可以看到，Δδ公式中模块度的计算和前面的δ公式不同，这里存在一个推导关系。实际上δ公式经过展开计算，就可以得到Δδ中括号里的表达式了。即：

$$\delta = \sum_{c}\left[\frac{\sum_{in}}{2m} - \left(\frac{\sum_{tot}}{2m}\right)^2\right]$$

这就和Δδ公式一致了。

7.3.3　用 GraphX 实现 Louvain 算法

了解了Louvain算法的原理，如何用GraphX来实现呢？因为GraphX内置了很多集合运算的操作，实现代码十分精简。这样一来，我们可以聚焦在算法实现上，而不用太考虑基础数据结构和基础操作的开发。

我们把项目起名为louvainTest（源代码可以在这里下载：https://github.com/Sotera/spark-distributed-louvain-modularity），它主要包含以下模块，如图7-4所示。

- LouvainCore.scala。算法的核心代码都在这里了。
- VertexState.scala。节点状态类，是Louvain算法的主要数据结构。
- Main.scala。主函数和一些参数配置项。
- HDFSLouvainRunner.scala。将临时结果写入HDFS的辅助类。
- LouvainHarness.scala。`HDFSLouvainRunner`的父类。

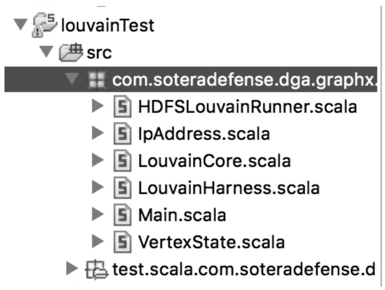


图7-4　louvainTest项目代码结构

前两个文件就是核心了。下面，我们逐步分析用GraphX实现Louvain算法的主要代码。

`main`函数主要代码如下：

```
/**
 * 读取数据文件，每行格式：id1 id2，表示id1和id2的关系好
 * 这里我们使用无向图
 *
 */
var edgeRDD = sc.textFile(edgeFile).map(row=> {
```

```
    val tokens = row.split(edgedelimiter).map(_.trim())
     tokens.length match {
         case 2 => {new Edge(inputHashFunc(tokens(0)),inputHashFunc(tokens(1)),1L) }
         case 3 => {new
             Edge(inputHashFunc(tokens(0)),inputHashFunc(tokens(1)),
             tokens(2).toLong)}
         //case _ => {throw new IllegalArgumentException("invalid input line: "+row)}
         case _ => { new Edge(0, 0, 1L)}
     }
})

println(s"total rows: "+edgeRDD.count())

// 根据配置项决定是否要把当前的RDD分片，并行处理
// 默认会根据HDFS存储的block数作为分片数
if (parallelism != -1 ) edgeRDD = edgeRDD.coalesce(parallelism,shuffle=true)

// 将RDD转换为图
val graph = Graph.fromEdges(edgeRDD, None)

// 这里使用HDFSLouvainRunner类运行核心算法并输出
// 如果不使用HDFS输出，可以通过继承LouvainHarness类实现
val runner = new HDFSLouvainRunner(minProgress,progressCounter,hdfspath,outputdir)
runner.run(sc, graph)
```

`main`的主要流程比较简单，从数据文件读取生成RDD后，转换为GraphX的Graph结构，再调用辅助类执行计算即可。其中，`run`函数就是计算模块度的主循环。下面我们看下`run`函数的实现：

```
def run[VD: ClassTag](sc:SparkContext,graph:Graph[VD,Long]) = {

var louvainGraph = LouvainCore.createLouvainGraph(graph)

var level = -1  // 统计图按社区压缩的次数
var q = -1.0    // 当前模块度的值
var halt = false
do {
    level += 1
    println(s"\nStarting Louvain level $level")

    // 调用Louvain算法计算每个节点所属的当前最佳社区
    val (currentQ,currentGraph,passes) = LouvainCore.louvain(sc,
        louvainGraph,minProgress,progressCounter)
    louvainGraph.unpersistVertices(blocking=false)
    louvainGraph=currentGraph

    saveLevel(sc,level,currentQ,louvainGraph)

    // 如果模块度增加量超过0.001，说明当前划分比前一次好，继续迭代
    // 如果计算当前社区的迭代次数少于3次，就停止循环
    //println(s"if ($passes > 2 && $currentQ > $q + 0.001 )")
    if (passes > 2 && currentQ > q + 0.001 ){
        q = currentQ
```

```
        louvainGraph = LouvainCore.compressGraph(louvainGraph)
    } else {
        halt = true
    }

} while ( !halt )

finalSave(sc,level,q,louvainGraph)
}
```

run函数主要分3部分，如下。

(1) 创建包含VertexState节点数据结构的Graph，通过LouvainCore.createLouvainGraph实现，这样才能继续下面的模块度计算。

(2) 计算每次迭代的模块度，判断是否已达最大并退出迭代：

   (a) 循环调用LouvainCore.louvain函数，计算每次的模块度；

   (b) 如果模块度比上一次迭代大0.001以上，则调用LouvainCore.compressGraph压缩当前图，重复步骤(2)；

   (c) 否则退出循环。

(3) 计算完成，保存结果。

循环中用到的参数（如模块度增加量和迭代次数限制）均可调整，在速度和效果间取得平衡就好。

下面就进入Louvain算法的核心代码，先来看计算使用的Graph结构LouvainCore.createLouvainGraph。

```
/**
 * 将RDD转换成Graph[VertexState,Long]类型的Graph，其中Long是权重类型，默认是1
 * Louvain算法将使用返回的Graph[VertexState,Long]进行计算
 */
def createLouvainGraph[VD: ClassTag](graph: Graph[VD,Long]) :
    Graph[VertexState,Long]= {
    // 计算每个目的节点的入度，作为权值（原算法是把出入度都相加）
    // (Long, Long)表示这个节点的（入度，出度）
    val nodeWeightMapFuncEx = (ctx:EdgeContext[VD, Long, (Long,Long)]) => {
        ctx.sendToDst((ctx.attr,0L))
        ctx.sendToSrc((0L,ctx.attr))
    }
    // 对一个节点的边信息做聚合。入度相加，出度相加
    val nodeWeightReduceFuncEx = (e1:(Long, Long),e2:(Long,Long)) => (e1._1+e2._1,
        e1._2+e2._2)

    val nodeWeights = graph.aggregateMessages(nodeWeightMapFuncEx,
        nodeWeightReduceFuncEx)

    val louvainGraph =
```

```
        graph.outerJoinVertices(nodeWeights)((vid,data,weightOption)=> {
        val weight = weightOption.getOrElse((0L, 0L))
        val state = new VertexState()
        state.community = vid // 将每个节点的社区ID初始化为自身ID
        state.changed = false
        state.communitySigmaTot = weight._1 // 入度
        state.internalWeight = weight._1
        state.nodeWeight = weight._2 // 出度
        state
    }).partitionBy(PartitionStrategy.EdgePartition2D).groupEdges(_+_)

    return louvainGraph
}
```

这里的主要工作就是初始化VertexState数据结构。

(1) 通过Graph.aggregateMessages分布式计算各个节点的出度、入度。得到中间结果集合nodeWeights: VertexRDD[(Long, Long)]。其中(Long, Long)代表该节点的(入度，出度)。

(2) 调用Graph.outerJoinVertices将graph: Graph[VD,Long]和nodeWeights:VertexRDD[(Long, Long)]关联计算：

   (a) 将每个节点的社区ID设置为自身ID，表示初始时每个节点作为一个社区；

   (b) nodeWeights: VertexRDD[(Long, Long)]中的节点出入度信息，填入VertexState结构。

(3) 返回新的Graph[VertexState,Long]图结构。

后续所有的计算都在Graph[VertexState,Long]上进行。下面，我们看下核心的LouvainCore.louvain函数。

```
/**
 * 在Graph[VertexState,Long]结构上，迭代计算每个节点所属的社区，直至模块度达到最大
 * 该函数中不包含对图的压缩，压缩在另一个函数中实现
 */
def louvain(sc:SparkContext, graph:Graph[VertexState,Long],minProgress:Int=1,
    progressCounter:Int=1):(Double,Graph[VertexState,Long],Int)= {
    var louvainGraph = graph.cache()
    val graphWeight = louvainGraph.vertices.values.map(vdata=>
        vdata.internalWeight+vdata.nodeWeight).reduce(_+_)
    var totalGraphWeight = sc.broadcast(graphWeight)
    println("totalEdgeWeight: "+totalGraphWeight.value)

    // 收集每个节点的邻居节点社区ID、社区权重并累加，为后面判断所属社区做准备
    var msgRDD = louvainGraph.aggregateMessages(sendMsgEx, mergeMsg).cache()
    var activeMessages = msgRDD.count() // Scala有延后执行的特性，这里为保证数据可用，
                                        // 强制执行

    var updated = 0L - minProgress
    var even = false
    var count = 0
```

```
val maxIter = 100000
var stop = 0
var updatedLastPhase = 0L

do {
    count += 1
    even = ! even

    // 计算当前节点所属的最佳社区
    val labeledVerts =
        louvainVertJoin(louvainGraph,msgRDD,totalGraphWeight,even).cache()

    // 统计所有社区的权重
    val communtiyUpdate = labeledVerts
        .map( {case (vid,vdata) =>
            (vdata.community,vdata.nodeWeight+vdata.internalWeight)})
        .reduceByKey(_+_).cache()

    // 将节点ID与新社区ID、权重关联
    val communityMapping = labeledVerts
        .map( {case (vid,vdata) => (vdata.community,vid)})
        .join(communtiyUpdate)
        .map({case (community,(vid,sigmaTot)) =>
            (vid,(community,sigmaTot)) }).cache()

    // 更新VertexState值，建立节点ID=>新VertexState映射
    val updatedVerts =
        labeledVerts.join(communityMapping).map({ case (vid,(vdata,
           communityTuple) ) =>
           vdata.community = communityTuple._1
           vdata.communitySigmaTot = communityTuple._2
           (vid,vdata)
    }).cache()
    updatedVerts.count()
    labeledVerts.unpersist(blocking = false)
    communtiyUpdate.unpersist(blocking=false)
    communityMapping.unpersist(blocking=false)

    // 得到更新后的Graph
    val prevG = louvainGraph
    louvainGraph =
        louvainGraph.outerJoinVertices(updatedVerts)((vid, old, newOpt) =>
            newOpt.getOrElse(old))
    louvainGraph.cache()

    // 更新msgRDD用于下次迭代，并将不用的临时表释放
    val oldMsgs = msgRDD
    msgRDD = louvainGraph.aggregateMessages(sendMsgEx, mergeMsg).cache()
    activeMessages = msgRDD.count()
    oldMsgs.unpersist(blocking=false)
    updatedVerts.unpersist(blocking=false)
    prevG.unpersistVertices(blocking=false)

    if (even) updated = 0
```

```
        updated = updated + louvainGraph.vertices.filter(_._2.changed).count
        if (!even) {
            println("  # vertices moved: "+
                java.text.NumberFormat.getInstance().format(updated))
            if (updated >= updatedLastPhase - minProgress) stop += 1
            updatedLastPhase = updated
        }
    } while ( stop <= progressCounter && (even ||  (updated > 0 && count < maxIter)))

    println("\nCompleted in "+count+" cycles")


    // 重新计算整体图的模块度
    val newVerts = louvainGraph.vertices.innerJoin(msgRDD)((vid,vdata,msgs)=> {
        val community = vdata.community
        var k_i_in = vdata.internalWeight
        var sigmaTot = vdata.communitySigmaTot.toDouble
        msgs.foreach({ case( (communityId,sigmaTotal),communityEdgeWeight ) =>
            if (vdata.community == communityId) k_i_in += communityEdgeWeight
        })

        val M = totalGraphWeight.value
        val k_i = vdata.nodeWeight + vdata.internalWeight
        // 计算新模块度
        var q = (k_i_in.toDouble / M) -  ( ( sigmaTot *k_i) / math.pow(M, 2) )
        //println(s"vid: $vid community: $community $q = ($k_i_in / $M) -
        //( ($sigmaTot * $k_i) / math.pow($M, 2) )")
        if (q < 0) 0 else q
    })

    val actualQ = newVerts.values.reduce(_+_)

    // 返回更新后的模块度和完整图
    return (actualQ,louvainGraph,count/2)
}
```

louvain函数比较长，基本上分为6部分。

(1) 在graph:Graph[VertexState,Long]上执行Graph.aggregateMessages，计算每个节点对其所有邻居社区的权重，得到msgRDD: VertexRDD[Map[(Long, Long), Long]]。这个RDD是个map，(社区ID，社区权重) $\overset{\text{映射到}}{\Rightarrow}$ 节点相对该社区的权重。

(2) 对graph:Graph[VertexState,Long]和msgRDD: VertexRDD[Map[(Long, Long), Long]]做关联，对每个节点计算其属于邻居社区的模块度Δδ，找到最大的Δδ，更新该节点所属的社区。得到labeledVerts: VertexRDD[VertexState]集合，记录了节点ID和对应的新VertexState。

(3) 将新社区信息更新到graph:Graph[VertexState,Long]上，分几步实现：

(a) 更新每个社区权值，在labeledVerts: VertexRDD[VertexState]上计算每个社区ID新权值，得到communtiyUpdate: RDD[(Long, Long)]集合，key是社区ID，

value是新社区权重；

(b) 将labeledVerts: VertexRDD[VertexState]和communtiyUpdate: RDD[(Long, Long)]关联，得到communityMapping: RDD[(VertexId, (Long, Long))]集合，key是节点ID，value是(社区ID, 新社区权重)；

(c) 将labeledVerts: VertexRDD[VertexState]和communityMapping: RDD[(VertexId, (Long, Long))]执行join操作，得到updatedVerts: RDD[(VertexId, VertexState)]集合，key是节点ID，value是包含了新社区ID和社区权重的VertexState；

(d) 最后将graph:Graph[VertexState,Long]和updatedVerts: RDD[(VertexId, VertexState)]执行outerJoinVertices操作，得到更新了本轮计算的新社区信息的louvainGraph: Graph[VertexState, Long]。

(4) 根据新的社区信息，更新msgRDD。

(5) 判断是否需要停止迭代，如继续迭代，则重复步骤(2)。

(6) 重新计算整个图的模块度，返回结果。

7.3.4　小试牛刀：谁是你的闺蜜

理解了GraphX实现的Louvain算法，我们用一个示例来测试下算法的效果。假设有下面几位同学，他们之间的关系互为好友，如图7-5所示。

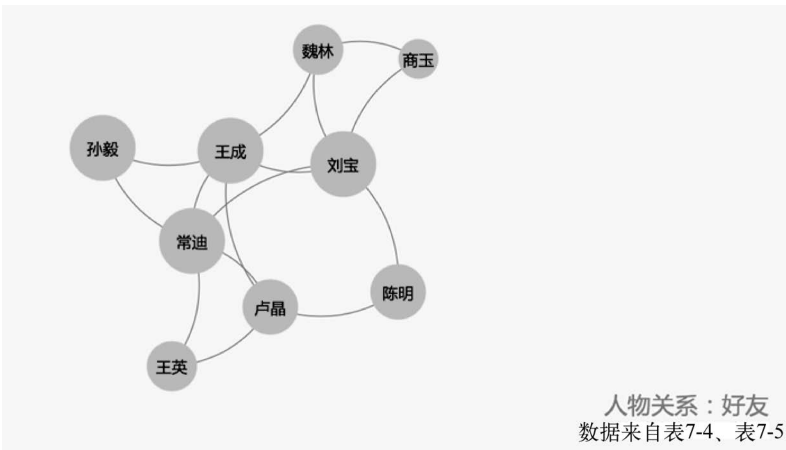


图7-5　好友关系示例

我们用数字代表他们的ID，参见表7-4。

表7-4　好友关系示例

ID	姓名	ID	姓名
1	常迪	6	陈明
2	刘宝	7	卢晶
3	王成	8	王英
4	孙毅	9	商玉
5	魏林		

他们之间的关系也用表来表示，如表7-5所示，“2 1”就表示刘宝和是常迪的好友。

表7-5　好友关系示例

学生ID	好友ID	学生ID	好友ID
2	1	7	1
3	1	8	1
3	2	8	7
2	3	1	7
1	3	3	7
4	3	5	2
4	1	9	5
1	4	3	5
6	2	9	2
2	6	1	2
7	6		

我们将这两列数字写入文件，作为Louvain算法的输入，放到Spark上运行，会得到如图7-6所示的一组结果。

```
15/12/29 20:35:23 WARN util.NativeCodeLoader: Unable to load native-hadoop library for
(6,{community:2,communitySigmaTot:11,internalWeight:0,nodeWeight:1})
(1,{community:3,communitySigmaTot:10,internalWeight:0,nodeWeight:4})
(7,{community:3,communitySigmaTot:10,internalWeight:0,nodeWeight:2})
(8,{community:3,communitySigmaTot:10,internalWeight:0,nodeWeight:2})
(2,{community:2,communitySigmaTot:11,internalWeight:0,nodeWeight:3})
(3,{community:2,communitySigmaTot:11,internalWeight:0,nodeWeight:4})
(9,{community:2,communitySigmaTot:11,internalWeight:0,nodeWeight:2})
(4,{community:3,communitySigmaTot:10,internalWeight:0,nodeWeight:2})
(5,{community:2,communitySigmaTot:11,internalWeight:0,nodeWeight:1})
```

图7-6　好友关系分析结果

如图7-6所示，第一列是各位同学的ID，第二列community就代表该同学的所属社区了。所以我们看到，刚才复杂的好友关系被分析出来两个社区。一个是以刘宝同学为核心的，成员有陈明（6）、王成（3）、商玉（9）、魏林（5）；另一个是以王成同学为核心的，成员有常迪（1）、卢晶（7）、王英（8）、孙毅（4）。

现在，你只要把你周围的朋友关系输入文件并调用Louvain算法，就可以轻松判断谁是你们之间的核心人物了。是不是很有趣?

7.3.5　真实的场景：新浪微博关系分析

看了上一节的示例，我们再用实际的数据看下效果。这里选取新浪微博2014年6月份的一部分用户收听关系数据（数据来源于网络），我们看下能否分析出谁是其中的热门人物。

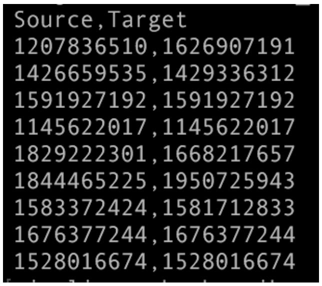
我们用1000万条的微博关注数据做验证（即这个图有1000万条边）。原始数据如图7-7所示，包括用户ID、注册信息、等级等，这里我们重点关注用户ID和关注列表，这两个字段能够体现用户之间的关系。



```
用户ID,屏幕名,性别,VIP描述,自我介绍,地区,用户名,关注,粉丝,微博,工作,教育,头像,加V,标签,生日,QQ,Msn,Email,创建时间,关注列表,会员,达人,等级
2314832262,阿佩佩Shop,女,,不定期上微博更新等, 看上的建議聯繫@Ylikann,广东 惠州,2314832262,104,174,37,,,http://tp3.sinaimg.cn/2314832262/180
249895040,2260374144,2918907995,2135678820,3737402632,2503355162,2619314083,2661503142,2181946944,2690991083,1774341190,1806754644,22530892
926450,3228043711,3665534963,3194082252,1965654147,2987324453,3282850914,2375446131,2646270043,1934329351,2359781860,1998519690,2203116877,
774,1941792707,3114311595,2606450563,2646526614,1777963820,2263538664,2425703402,2681129613,1741625184,1978929364,1876956375,2182739860,270
,2255748934,2716113345,3032350180,1926176395,2652166882,2600258430,3313897775,2831522324,2732627387,2663871245,2929336851,2122832790,178800
56783950,2702316587,2155261370,2861083622,1625321080,3220614835,2535081194,1882258243,3625514562,3514017581,2840400877,2840959340,313678505
72834,2683494454,3611273833,1795651861,1934167143,2698830971,2859236282,1916809075,1930585993,2607985647,2403228864,2299596367,2297157182,2
54,3101406441,3080153381,2637240063",0,0,6
```

图7-7　原始微博数据示例

当然，这样的原始数据是无法直接分析的，我们需要做解析和处理，最后得到的数据格式如图7-8所示。



```
Source,Target
1207836510,1626907191
1426659535,1429336312
1591927192,1591927192
1145622017,1145622017
1829222301,1668217657
1844465225,1950725943
1583372424,1581712833
1676377244,1676377244
1528016674,1528016674
```

图7-8　　格式化后的微博数据示例

如图7-8所示，第一列是用户ID，第二列是被关注者的ID。也就是说，第一列是图中的源节点，第二列是目标节点，这样一行就代表了微博上的一条关注关系。

我们的运行环境是3台Xeon E5606（4核，32 GB内存）的Spark集群，分别对1000万条和3000万条关注关系数据执行Louvain算法。性能对比如表7-6所示。

表7-6　数据分析性能对比

数　据　量		运行时间
边数：1000万	节点数：405万	10分钟
边数：3000万	节点数：1065万	54分钟

Louvain算法实际是$O(n^2)$的时间复杂度。可以看出，随着图规模的扩大，计算量的增长更加明显，其间并非线性关系。

以1000万数据样本为例，我们共分析得到100个主要社区（这里把成员数量超过500的社区算为主要社区），形成一个如图7-9所示的社区分布图。

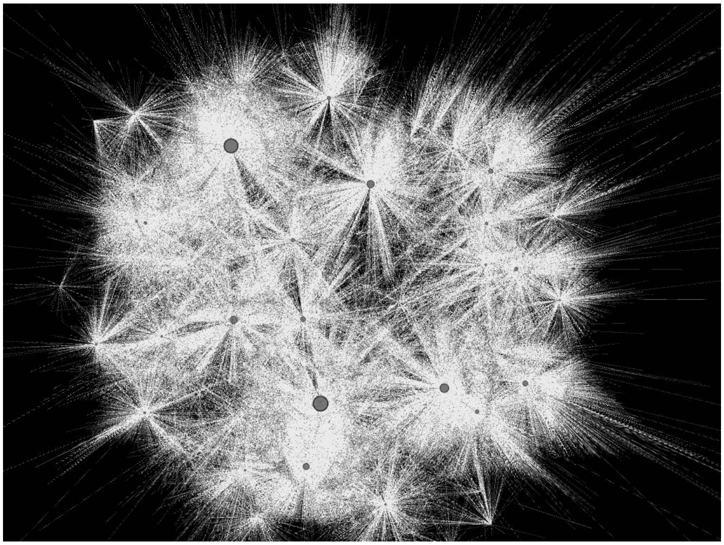


图7-9　1000万微博关系数据分析结果可视化展现

该图用Gephi软件生成。因为数据量比较大，网上通常的可视化库如ECharts等效果都不理想，还是Gephi比较强大，跨平台、开源，是一个做SNA理想的可视化分析工具。

我们来看看这些社区是什么。先截取圈子成员最多的社区微博ID，如图7-10所示。

```
Count WeiboID
  76803 5128778185
  71885 5096677199
  44023 5147135536
  36671 5149153146
  36609 5136075296
  31278 5102661437
  25280 5149628707
  24845 5142589487
  22190 5127036370
  18853 5088164744
  17707 5108511555
  17564 5157873723
  17341 5051267375
  13961 5143855901
  13346 5121404532
  11107 3953943931
  10783 5138067801
  10263 5151317833
   9987 5099507225
```

图7-10　社区排名示例

如图7-10所示，第一列是社区的成员数量，第二列是社区核心的微博ID。因数据是2014年的，部分ID可能已失效，部分ID的数据也会不准。但我们仍能从排名靠前的社区里看到一些有趣的聚集。

比如，其中一个社区是卖婴幼儿产品的官微，如图7-11所示。



图7-11　社区示例

我们还能找到某著名演员的粉丝社区，如图7-12所示。



图7-12　社区示例

我们来看看她的粉丝社区是什么样的。在抽样的1000万数据中，直接关注她的粉丝数量（如图7-13所示，核心的节点即是该演员的微博ID）大概100多个。但经过Louvain算法分析后，我们再来看看她的粉丝社区。如图7-14所示，这时我们发现其粉丝社区的成员数量达到1万个以上，

扩大了100倍！这也说明了她关系链的影响力比较大，当她发布一条信息时，可以通过其关系链影响力，成百倍的扩大传播范围。

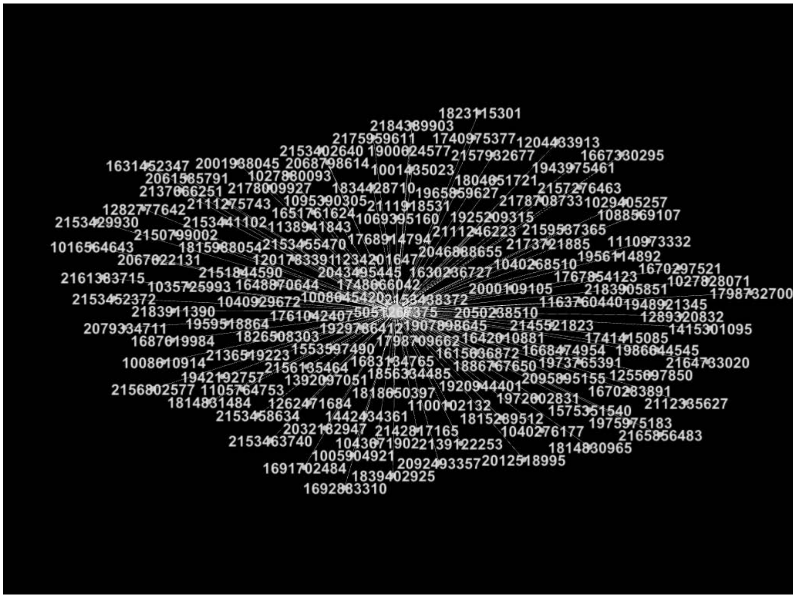


图7-13　粉丝社区可视化示例

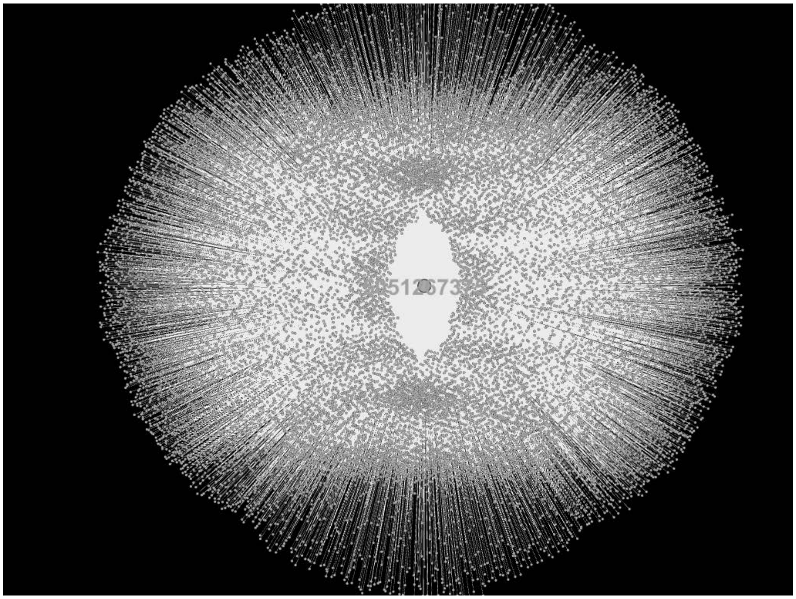


图7-14　社区分析结果可视化示例

最后，我们看看她的社区里都有哪些类型的粉丝，如图7-15所示。



```
3261692223      PoohJeter       体育
3028617833      、偶是好偶人    健康
3221711201      RoseInLife      健康
3259147044      愛妳瞬  健康
3259154110      晓月巫山云      健康
2671925027      罗嗦罗嗦的爱    娱乐
3191671495      2013湖南卫视元宵喜乐会  娱乐
3158246263      恨ni_让wo如此shang      文艺
2833862400      灿、若信仰      旅游
3153979080      善良1234hu      旅游
3174042654      海上孤塔        旅游
2694275643      豆豆冰山-期待北外       时尚
2764955005      Cyuan1987       汽车
2921220312      金昌交警田军    汽车
2940429325      wy粉末  美食
2958856855      黄金满地一叶知秋        美食
2875396815      冬-花-花        微博奇葩
3033822323      幸福玉姐1990    投资理财
3178557202      少年新的起航2013        教育就业
2810150734      喂-阿威 新闻趣事
2242661672      nature-陈小然   星座命理
3228252993      我叫MT口袋魔兽  游戏动漫
3214257523      心碎_流浪猫     视频音乐
```

图7-15　粉丝社区分析示例

如图7-15所示，第一列是微博ID，第二列是昵称，第三列则是该用户的标签。从标签上来分析，健康、文艺娱乐和旅游是这个社区的3个主要特点。

到这里，我们的社交网络分析之旅就告一段落了。通过Spark + GraphX + Louvain算法的组合，我们可以轻松地将这套方法应用到其他分析场景上。开发人员只需关注具体分析算法，基础的图运算库、数据量、扩容／容灾能力、性能都可以交给Spark + GraphX框架处理，这是最大的优势所在。

第 8 章 Spark MLlib

MLlib是Spark为解决机器学习问题开发的库，这些问题包括：分类、回归、聚类、协同过滤等。另外，Spark从1.2版本开始为搭建机器学习Pipeline系统提供接口：`org.apache.spark.ml.Pipeline`。Pipeline API为不同训练数据的处理、特征选择的处理，以及参数调优提供了便利。从本质上看，MLlib就是RDD上一系列可供调用的函数的集合。

8.1 机器学习简介

开始介绍MLlib前，我们先简单介绍一下机器学习。机器学习涉及的知识面比较广，本书并不打算做太多深入探讨，感兴趣的读者可以查看8.1.5节的参考资料。当然，熟悉这块的读者也可以略过这一节。

8.1.1 什么是机器学习

什么是机器学习？Herbert Sinmon给“学习”做出了这样的定义：“如果一个系统能够通过执行某个过程而改进性能，这就是学习。”更通俗的理解就是：机器学习能够自动地从数据中学习“程序”，而这个“程序”不是人来编写的。我们来看一个简单的例子。

平面上有两类点：黄色点表示类别a，蓝色点表示类别b（参见图8-1）。这时我们希望能够找到平面上的一条曲线L，将两个类别的点分成两个空间，使类别a属于空间A，类别b属于空间B。这样一来，对于一个新出现的颜色未知的点X，我们通过查看点X落在空间A还是空间B中来判断X属于哪个类别。

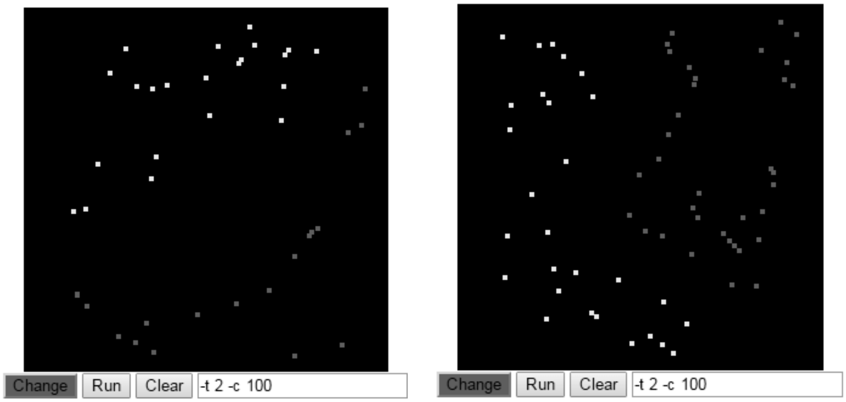


图8-1　SVM算法效果演示：学习的对象（另见彩插图8-1）

当然，这个任务从人的视觉上来看是比较容易完成的。人也可以将这个曲线L用一个数学表达式（比如二次方程）来表示，但这个曲线方程是确定的，当观察到的数据发生变化时，需要重新调整方程式，因此扩展性不好。机器能够根据已经观测到两个类别不同的情况自动给出不同的曲线L的具体表达式，这里的L就是我们常说的学习到的模型（参见图8-2）。这是一个有监督学习的典型示例。

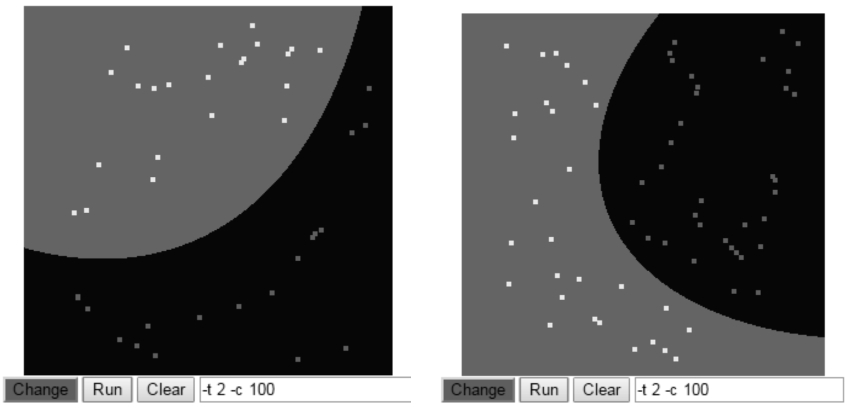


图8-2　SVM算法效果演示：学习的效果（另见彩插图8-2）

这个例子中自动寻找曲线L（分界面）的算法是由SVM程序完成的，参见http://www.csie.ntu.edu.tw/~cjlin/libsvm/。更多内容请参考关于支持向量机（support vector machine）的其他文献。

如今，机器学习已经被广泛应用于各个领域，举例如下。

- **图像**。人脸识别任务，可以识别图像中人脸的位置。
- **声音**。将声音转换成文字任务，著名的应用有苹果的Siri。
- **文本**。拼写纠错，搜索引擎大量应用文本挖掘。

8.1.2 机器学习示例

本节我们再通过一个小例子深入了解下机器学习及其相关概念。

这个例子旨在根据多次测量估计腾讯大厦的精确高度x（参见图8-3）。



图8-3 腾讯大厦

测量的具体过程是将卷尺从顶楼抛到地面来读数测量。由于大楼表面凹凸不平，以及风力等原因，每次读到的数据都不太一样：193.3 m、193.7 m、192.8 m……那么，大厦精确的高度应该是多少呢？假设实际高度是193.2 m，我们称实际高度为理论值。我们可能永远不知道理论值具体是多少，但可以让估计值尽量接近这个理论值。

最小二乘法就是一个这样的理论。它定义多次测量值的误差之和为累积误差：

$$累积误差 = \sum（观测值-理论值）^2$$

我们用$g(x)$表示其累积误差，x表示理论值的估计值，x_i表示第i次测量，得到数学表达式：

$$g(x)=\sum_{i=1}^{n}(x-x_i)^2$$

这里x_i为已知值，依据最小二乘法可知，当累积误差最小时，我们就得到一个最接近理论值的估计值。这是求函数极值的常见问题。我们对x求导，得到公式：

$$g'(x)\big|_x=\sum_{i=1}^{n}2(x-x_i)=2(nx-\sum_{i=1}^{n}x_i) \qquad (8.1)$$

导数$g(x)'$为0时，即$g(x)'=0$，$g(x)$取极小值，我们令公式8.1等于0，求解方程式得到：

$$x=\frac{1}{n}\sum_{i=1}^{n}x_i$$

估计值x等于各次观测值x_i的平均值。

所以，我们日常中使用多次测量，然后对多次测量求平均值的做法是有严谨数学依据的。当然，这个例子比较简单，在机器学习的其他问题中，问题的抽象和求解可能要比这里复杂得多。

8.1.3　机器学习的基本方法

机器学习已经形成了一套完善的方法论，在很多文献中“机器学习方法 = 模型 + 策略 + 算法”。在上一节的例子中，我们介绍了最小二乘法，这里将这个例子和机器学习的基本方法做一个简单的对应，如下。

- **模型**（model）。计算机如何表达要解决的问题，就是机器学习的模型，这里使用一个函数$f(x_i)$来表达，具体函数实现是为求均值。
- **策略**（strategy）。模型通常是有参数的，所以可能的模型有很多个，如何评估模型的优劣就是机器学习的策略，这里通过计算误差平方和评估模型的优劣，这个误差平方和通常叫作平方损失函数。
- **算法**（algorithm）。损失函数评估模型的优劣通常通过一个搜索算法来找到最优的模型，这里通过函数求导来搜索损失函数值最小的算法，借此来选择最优模型。

机器学习的方法丰富多样，模型可以是函数$f(x)$，常见的还有概率分布$P(y|x)$；策略可以是平方和最小，常见的损失函数（loss function）还有0-1损失函数、绝对损失函数、对数损失函数等；算法可以是求导，也可以是EM算法等。

机器学习的一般过程大概可以分为几个步骤：

(1) 准备训练数据集合；

(2) 选择学习模型；

(3) 选择学习策略；

(4) 实现求解最优模型的算法；

(5) 使用最优模型对新数据进行预测。

整个过程可以用图8-4表示。

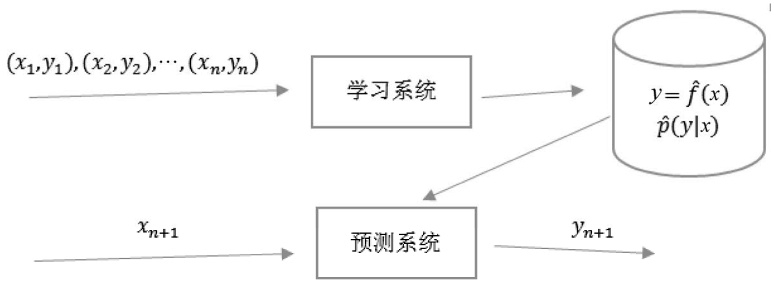


图8-4　机器学习的一般过程

8.1.4 机器学习的常见技巧

上一节介绍了机器学习的基本方法和一般步骤，但是实际应用过程中，每一个步骤都有很多细致的工作要做。

首先，在准备训练数据时，训练数据要注意满足以下两点：

- 尽可能和应用场景同分布；
- 尽可能充分，而且充足。

为了防止过拟合和模型验证，通常我们会把训练数据分为训练集合、验证集合和测试集合：训练集合用于模型训练，验证集合用于模型选择，测试集合用于模型评估。

选择好训练样本以后，我们需要将其通过特征向量表示成算法能够理解的形式。特征的数值类型表示形式如下。

- **浮点数特征**。比如商品的价格。
- **离散值特征**。比如商品的颜色，用每个数字表示一种颜色。
- **二值特征**。例如，一个广告标题中是否有“打折”这个词作为特征，有则特征取值为`1`，没有则取值为`0`。

其中，直接使用浮点数特征时，如果不同特征取值范围差异很大，可能导致学习的模型不是最优解。所以我们需要把不同特征的取值都映射到相同的范围内，通常通过归一化（normalization）方法进行缩放，还需要处理缺失值（missing value）和异常值，异常值比如本章案例中需要过滤点击数大于曝光数的样本。当特征空间比较大时，我们需要做特征选取（feature selection）。这时，我们应选择信息增益大的特征，否则模型会太大，训练和预测效率都会很低。特征选择上，一方面不是特征越多效果越好，特征之间可能携带的信息有重合；另一方面则需要保证特征包含足够区分需要识别的对象的信息。业界的一些系统已经能够做到部分机器自动选取特征的过程。

选择模型上，不是越复杂的模型越好，也没有万能的模型，往往诠释要解决的问题本身比理解模型更难。模型选择一般还要考虑以下几点。

- **训练时间**。随着观测数据的变化，需要重新训练模型，训练新模型的时间不能太长。
- **预测时间**。模型上线工作的时候，对于新的输入预测得分所需要的时间。
- **模型的存储**。模型运行的时候需要多少内存空间。

能够理解和调试选择的算法，对于理解模型和优化效果也有很大帮助。

一般模型需要多次调优才能够取得较好的效果，这时候分析案例往往能够洞察你的问题。比如，若发现预测效果不好的样本普遍存在某个规律，通常我们会从细分的各种维度去分析模型的效果，来确定下一步的优化方向。

实际工程应用中，准备数据和特征往往占据大比例工作量。

8.1.5　机器学习参考资料

前面我们通过简单例子对机器学习理论和方法进行了简单介绍。机器学习涉及的知识面较广，包括了数理统计、数值优化、线性代数等。这里介绍一些资料，供感兴趣的朋友进一步学习。

首先要推荐李航的《统计学习方法》，本书通俗易懂，系统地介绍了机器学习的常见算法。另外，书中还提供不少例子，通过手工计算例子读者能够较快地掌握算法。

此外，机器学习大牛Andrew Ng的视频课程*Machine Learning*是目前的高校经典课程，其中也包括不少参考资料和练习题。具体参见：

http://openclassroom.stanford.edu/MainFolder/CoursePage.php?course=MachineLearning

网易公开课做了一个中文字幕的版本：

http://v.163.com/special/opencourse/machinelearning.html

Christpher M. Bishop的著作*Pattern Recognition And Machine Learning*是模式识别与机器学习的经典教材。该书每章都有相应的习题及答案，深入浅出地介绍了模式识别与机器学习的基本理论和主要方法。同时，本书还涵盖了模式识别与机器学习领域的一些最新进展，不仅适合初学者学习，而且对专业研究人员也有很大的参考价值。

Toby Segaran的《集体智慧编程》和Peter Harrington的《机器学习实战》也是比较好的入门书。这两本书的特点是没有深奥的数学原理，而是通过“问题实例+实际代码+运行效果”来介绍机器学习算法，代码采用Python语言编写，十分便于初学者调试算法。

此外，Pedro Domingos的文章“A Few Useful Things to Know about Machine Learning”也是经典，对于机器学习应用实践提供了很多建议。

8.2　MLlib 库简介

接下来我们先简单介绍下MLlib库中基础的数据结构和常用的算法，后续的案例中会用到。

8.2.1　基础数据类型

为了便于理解后续的案例，这里介绍几个常用的基础数据类型。

1. 向量

向量（vector）通过`import mlib.linalg.Vectors`来使用，MLlib支持稠密向量和稀疏向量。稀疏向量只存储值非零的项，比如，声明`Vectors.sparse(1095023,Array(1,7,31),Array(1.0,1.0,0.0))`，其中`1095023`表示最大的下标，第一个数组表示非零值的下标，第二个数组表示非零项的具体值。

2. `labeled point`

`labeled point`是一个带标注（label）的向量，也可以是稠密或者稀疏的，用于监督学习

中表示一个特征向量和一个标注。这里规定标注是double类型。对于二分类来说，标注可以是0（负例）或者1（正例）。

3. 各种模型类

训练算法输出就是各种模型（model）类，本章案例使用的是org.apache.spark.mllib.classification.LogisticRegressionModel，常用的接口如下。

- **LogisticRegressionModel.load()**。加载一个训练好的模型；
- **LogisticRegressionModel.predict()**。对输入进行一次预估，输入参数是一个向量组成的RDD；
- **LogisticRegressionModel.save()**。将训练好的模型保存到磁盘，保存的模型通过load加载。

另外，在MLlib算法的实现过程中，代码中向量计算频繁出现Breeze相关的函数。Breeze是矩阵（matrix）及向量（vector）的计算库。Breeze是ScalaNLP中的项目，具体参考www.scalanlp.org。

8.2.2 主要的库

MLlib库主要有这几块：分类、回归、聚类、推荐等（参见表8-1）。

表8-1 主要的类库

主要的类库	说　　明
mllib.classification	分类算法，包括二分类和多分类，实现了逻辑回归、朴素贝叶斯、SVM等算法，常用的类有：LogisticRegressionWithLBFGS、LogisticRegressionWithSGD、NaiveBayes、SVMWithSGD
mllib.clustering	聚类，实现了K-Means、LatentDirichletAllocation算法，支持KMeans、LDA
mllib.recommendation	主要使用协同过滤方法做推荐，实现了Alternating Least Squares算法，常用的类有ALS
mllib.regression	回归算法，常用的类有LinearRegressionWithSGD、RidgeRegressionWithSGD、LassoWithSGD
mllib.tree	实现了决策树和随机森林等算法，常用的类有DecisionTree、GradientBoostedTrees、RandomForest

广告是互联网公司的主要利润来源之一，本章将介绍一个广告点击率预估的场景。此处我们主要使用其中分类算法中的逻辑回归，具体实现有LogisticRegressionWithLBFGS和LogisticRegressionWithSGD，两种实现在模型和学习策略上是一样的，只是学习算法不一样，LBFGS训练速度更快。

除了这些核心的算法，还有一些辅助处理的模块，参见表8-2。

表8-2 其他常用的类库

其他常用类库	说　　明
mllib.stat	基础统计，常用的有Statistics类，计算最小值、最大值、均值、方差等
mllib.feature	特征处理，有用于文本处理的HashingTF类和IDF类，以及用于数值归一化的StandardScaler类

（续）

其他常用类库	说　明
mllib.evaluation	算法效果衡量，常用的有BinaryClassificationMetrics类、MulticlassMetrics类、RegressionMetrics类等
mllib.linalg	基础线性代数运算支持，支持奇异值分解、降维等

MLlib库具体每个接口的参数说明这里就不展开叙述了，读者可以参考官方文档。Spark的官方文档建设得比较齐全，同时配备了详细的示例程序，一般都有Python、Java和Scala的版本。

另外，了解当前有哪些函数，一个便捷的办法就是在spark-shell下输入improt org.apache.spark.mllib.xxx.（查询xxx的类名下面有多少接口，参见图8-5），然后按TAB键。

```
scala> import org.apache.spark.mllib.regression.
GeneralizedLinearAlgorithm         GeneralizedLinearModel             IsotonicRegression
IsotonicRegressionModel            LabeledPoint                       LabeledPointBeanInfo
LassoModel                         LassoWithSGD                       LinearRegressionModel
LinearRegressionWithSGD            RegressionModel                    RidgeRegressionModel
RidgeRegressionWithSGD             StreamingLinearAlgorithm           StreamingLinearRegressionWithSGD
impl
```

图8-5　mllib.regression接口示例

8.2.3　附带的示例程序

Spark安装包里面附带了大量的示例数据和示例程序，这些文档和程序是学习使用Spark最有效的办法和手段。这里以Spark 1.4为例，做一个简单介绍。

示例数据在这个目录：

```
shifei@shifei-vm:~/spark-1.4/data/mllib$ ls
als                                      sample_isotonic_regression_data.txt
gmm_data.txt                             sample_lda_data.txt
kmeans_data.txt                          sample_libsvm_data.txt
lr-data                                  sample_linear_regression_data.txt
lr_data.txt                              sample_movielens_data.txt
pagerank_data.txt                        sample_multiclass_classification_data.txt
ridge-data                               sample_naive_bayes_data.txt
sample_binary_classification_data.txt    sample_svm_data.txt
sample_fpgrowth.txt                      sample_tree_data.csv
```

示例代码有不同的语言版本：

```
shifei@shifei-vm:~/spark-1.4/examples/src/main$ ls
java  python  r  resources  scala
```

本地安装Spark以后，运行示例程序十分便利，具体安装前请参考前面的章节。

比如，Spark文档中提供了一个线性回归的示例（代码如下）。我们简单修改了数据路径前缀以后，直接贴到spark-shell的命令行里面就可以运行了。

```
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.regression.LinearRegressionModel
import org.apache.spark.mllib.regression.LinearRegressionWithSGD
import org.apache.spark.mllib.linalg.Vectors

// 加载和解析数据
val data =
    **sc.textFile("/home/shifei/spark-1.4/data/mllib/ridge-data/lpsa.data")**
val parsedData = data.map { line =>;
    val parts = line.split(',')
    LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ').
        map(_.toDouble)))
}.cache()

// 训练模型
val numIterations = 100
val model = LinearRegressionWithSGD.train(parsedData, numIterations)

// 使用模型进行预估，并评估模型
val valuesAndPreds = parsedData.map { point =>;
  val prediction = model.predict(point.features)
  (point.label, prediction)
}
val MSE = valuesAndPreds.map{case(v, p) =>; math.pow((v - p), 2)}.mean()
println("training Mean Squared Error = " + MSE)
```

图8-6是在spark-shell粘贴代码以后的运行结果，结果的最后几行是：

```
scala> println("training Mean Squared Error = " + MSE)
training Mean Squared Error = 6.207129722351617
```

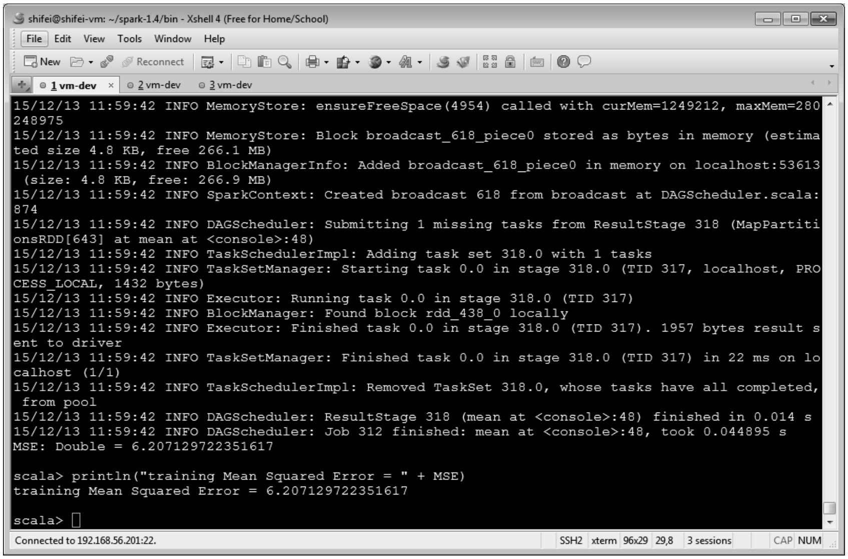


图8-6　spark-shell运行效果

均方误差（Mean Squared Error，MSE）是评估模型的常用指标，用来衡量“平均误差”。还有一些常用指标，我们将在后面的章节中进行介绍。

大部分例子都比较简单，接下来我们将基于应用场景编写一些复杂的程序。

8.3　应用场景：搜索广告点击率预估系统

点击率预估是指系统去预估用户可能点击某个内容的概率，这个内容可以是推荐的商品，也可以是广告。预估基本方法通常基于用户历史行为日志进行建模，这个建模过程随着日志的更新，需要重复地进行。互联网广告是互联网公司的主要收入来源之一，广告的点击率预估也是各个公司的核心技术。接下来就让我们在Spark系统上，利用机器学习的方法，来开发一个广告点击率预估系统。

8.3.1　应用场景

当用户在搜索引擎中输入一个查询词，搜索引擎除了返回相关的网页，同时还会返回一些广告（参见图8-7），这就是大家熟知的搜索关键词广告，谷歌的Adwords和百度的凤巢主要采用这一形式。这时，我们需要预估针对某个用户的查询词、点击某个广告的概率，这时用户（`user_id`）和查询串（`query_id`）是输入。



图8-7　百度搜索“榴莲”的搜索结果

搜索引擎将每次用户的搜索行为定位为一次会话（session），每天大量的会话被记录下来形成了搜索日志，这些日志作为预估下一次用户搜索的主要依据使用。

举个例子。某一次会话中，用户A输入查询词“芒果蛋糕”，并点击了蛋糕广告商B的广告C。这个“芒果蛋糕”就是一个query，query经过搜索引擎的query分析模块处理以后，以多个token

形式存在，被处理成token“芒果”和token“蛋糕”。同时A、B、C的相关属性也会被记录到日志中。这些日志被用作依据来建立点击率预估模型，这样的好处在于：

(1) 当出现一个新的query“榴莲蛋糕”时，也能够有较相关的广告展示；

(2) 当出现一个新用户的时候，也输入类似的“芒果蛋糕”时，和广告C类似的广告有较高的点击率预估值。

8.3.2　逻辑回归

8.2节介绍了机器学习的一般方法，我们知道机器学习方法由“模型 + 策略 + 算法”构成，这里我们使用同样的方法来解决8.3.1节提到的问题。

首先来讨论一下这里的模型：点击模型。

一个广告展现以后是否被点击，只有两种情况：“被点击”和“不被点击”。可以这样描述点击模型：

- 用模型预测值为1来表示“被点击”；
- 用模型预测值为0来表示“不被点击”。

所以，点击模型是一个二元分类器：$P(c)$, $c=\{0,1\}$。

实际上，我们很难准确地预估一定会点击，所以往往预估被点击的概率为P，P介于0和1之间。点击率预估模型就是用来预估被点击的概率，这样我们就希望模型输出一个介于0和1之间的数。

Logistic函数值域正好满足0~1这样一个特征，它的一个函数图形如图8-8所示。

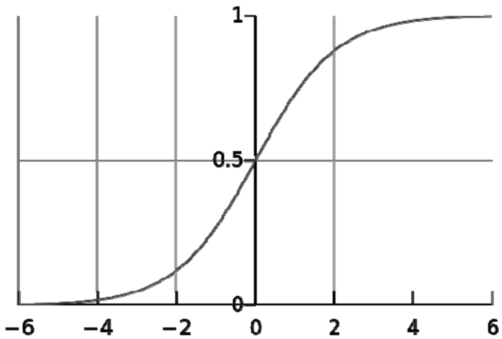


图8-8　Logistic函数图形

Logistic函数公式如下：

$$P(t)=\operatorname{logit}^{-1}(t)\equiv\frac{1}{(1+e^{-t})}=\frac{e^{t}}{1+e^{t}}$$

它是logit函数的反函数。logit函数定义如下：

$$\text{logit}(p)=\log\left(\frac{p}{1-p}\right)$$

于是，一个广告“被点击”的概率可以表示为：

$$P(c_i=1\mid x_i)=\frac{1}{1+\mathrm{e}^{-\omega\cdot x_i}}=\log\text{it}^{-1}(\omega\cdot x_i)$$

“不被点击”的概率可以表示为：

$$P(c_i=0\mid x_i)=1-\log\text{it}^{-1}(\omega\cdot x_i)$$

这里ω是模型的参数，也常写成向量的形式ω^t。x_i是模型的特征变量，每次广告展现时，特征变量动态计算出来，是已知的。这里模型的参数是未知的，我们可以从日志里面记录的信息去估计模型的参数ω。我们希望参数的取值能够使模型最接近观测到的日志数据，既观测事件发生的概率最大，这个方法叫**极大似然估计**（maximum likelihood estimation）。似然函数表示为：

$$p_i^{c_i}\cdot(1-p_i)^{1-c_i}\text{，其中 } p_i=\frac{1}{1+\mathrm{e}^{-\omega\cdot x_i}}$$

极大似然估计可以表示为似然函数值最大时，变量ω的取值，表示为：

$$argmax_{\omega}\prod\nolimits_1^n p_i^{c_i}*(1-p_i)^{1-c_i} \qquad (8.2)$$

将求似然函数最大值（公式8.2）作为目标，这就是机器学习三要素里面的策略（strategy）。

极大似然估计是求公式8.2极值，一般我们把问题转化成求对数似然函数。因为log函数是一个单调递增函数，极大似然函数就转化成求极大对数似然函数的问题。所以，公式8.2转化成对数似然函数：

$$\begin{aligned}L(\omega)&=\sum_{i=1}^{N}[c_i\log(P(c_i=1\mid x_i))+(1-c_i)\log(1-P(c_i=1\mid x_i))]\\&=\sum_{i=1}^{N}\left[c_i\log\left(\frac{P(c_i=1\mid x_i)}{1-P(c_i=1\mid x_i)}\right)+\log(1-P(c_i=1\mid x_i))\right]\\&=\sum_{i=1}^{N}\left[c_i(\omega\cdot x_i)-\log(1+\exp(\omega\cdot x_i))\right]\end{aligned} \qquad (8.3)$$

我们定义负对数似然函数：$l(\omega)=-L(\omega)$，将极大对数似然函数取负数，把求最大极值的过程转化成求最小极值的问题，负对数似然函数$l(\omega)$也就是8.1节提到的损失函数（loss function），通常叫对数似然损失函数(log-likelihood loss funciton)。这个对数似然损失函数是一个光滑的凸函数，

因此可以使用凸优化的一些方法来求解ω的估计值。

很多文献上直接给出的损失函数的形式：

$$L(\omega)=\log(1+\exp(-y\omega\cdot x_i))$$

这里使用了一些技巧，通过重新定义一个新函数来简化公式，这个函数定义是根据c_i的不同取值，y取值1或–1，分两种情况：

$$\begin{aligned} c_i=1\text{时，}\ l(\omega) &= -\log(P(y=1|x)) \\ &= -\log(1/(1+\exp(-\omega\cdot x))) \\ &= \log(1+\exp(-\omega\cdot x)) \end{aligned}$$

$$\begin{aligned} c_i=0\text{时，}\ l(\omega) &= -\log(P(y=-1|x)) \\ &= -\log(\exp(-\omega\cdot x)/(1+\exp(-\omega\cdot x))) \\ &= \log((1+\exp(-\omega\cdot x))/\exp(-\omega\cdot x)) \\ &= \log(1+\exp(\omega\cdot x)) \end{aligned}$$

这个公式和公式8.3是等价的。

可以这样直观地理解Logistic Loss：当$y=1$时，$\omega\cdot x$越大，Loss越小；当$y=-1$时，$\omega\cdot x$越小，Loss越小。

另外，我们常说逻辑回归模型是一个线性模型，是因为当拿被点击概率除以不被点击概率时，会得到一个线性的式子：

$$\log\left(P(c_i=1|x_i)/(1-P(c_i=1|x_i))\right)=\omega\cdot x$$

被点击概率和不被点击概率的比值也叫作odds，所以很多文献也称这个式子为log odds ratio。

根据logit函数定义，log odds ratio也可以使用logit函数表示为：

$$\text{logit}\left(P(c_i=1|x_i)\right)=\omega\cdot x$$

8.3.3 学习算法

上一节提到，我们选择学习的模型是逻辑回归模型，选择的学习策略是极大化对数似然函数的值，下来来讨论下学习算法，看看如何求损失函数的最小值，既负对数似然函数的极小值。梯度下降（gradient descent）是求解这一类无约束最优化问题的一种常见办法。梯度下降是一种迭代算法，通过选取合适的初值，不断迭代更新公式8.3中的ω，使目标函数极小化，直到收敛。负梯度是是函数下降最快的方向，每一步以负梯度方向更新ω值，从而达到减小目标函数值的目的。

这里，关键的过程就是每一步求解目标函数的梯度，为了加快收敛速度，以及梯度计算存储

和计算量的优化，实践中通常使用优化后的算法L-BFGS（Limited-memory BFGS）。L-BFGS是一种拟牛顿法（Quasi-Newton method），通过存储最近*m*轮的迭代信息，然后再通过这些计算梯度优化了存储量。

另外，根据训练集构成和单次梯度计算使用的样本数，又可以如表8-3这样划分。

表8-3　算法对比

	batch	mini-batch	Stochastic	Online
训练集	固定	固定	固定	实时更新
单次梯度计算	整个训练集	训练集的子集	单个样本	单个样本

当然，我们使用Spark的好处就在于不用考虑这些数据并行和计算并行的问题。

MLlib提供的`LogisticRegressionWithSGD`就是随机梯度下降（stochastic gradient descent）的一种实现，`LogisticRegressionWithLBFGS`是拟牛顿法的一种实现。两个算法的主要实现分别在`runMiniBatchSGD`和`runLBFGS`两个函数中，核心代码如下：

```
object GradientDescent extends Logging {
    /**
     * SGD算法
     */
    def runMiniBatchSGD(
        data: RDD[(Double, Vector)],
        gradient: Gradient,
        updater: Updater,
        stepSize: Double,
        numIterations: Int,
        regParam: Double,
        miniBatchFraction: Double,
        initialWeights: Vector): (Vector, Array[Double]) = {

            val stochasticLossHistory = new ArrayBuffer[Double](numIterations)

            val numExamples = data.count()

            if (numExamples == 0) {
                logWarning("GradientDescent.runMiniBatchSGD returning initial
                    weights,no data found")
                return (initialWeights, stochasticLossHistory.toArray)
            }

            if (numExamples * miniBatchFraction < 1) {
                logWarning("The miniBatchFraction is too small")
            }

            var weights = Vectors.dense(initialWeights.toArray)
            val n = weights.size

            var regVal = updater.compute(
                weights, Vectors.dense(new Array[Double](weights.size)), 0, 1,
```

```
                regParam)._2

        for (i <- 1 to numIterations) {
            val bcWeights = data.context.broadcast(weights)
            // mini batches方法，采样一部分数据集，并计算对应的梯度和
            val (gradientSum, lossSum, miniBatchSize) = data.sample(false,
                miniBatchFraction, 42 + i)
                .treeAggregate((BDV.zeros[Double](n), 0.0, 0L))(
                seqOp = (c, v) =>; {
                     // c变量保存的内容是(grad, loss, count), v变量保存的内容是 (label,
                     // features)
                     val l = gradient.compute(v._2, v._1, bcWeights.value,
                         Vectors.fromBreeze(c._1))
                     (c._1, c._2 + l, c._3 + 1)
                },
                combOp = (c1, c2) =>; {
                    // c变量保存的内容是(grad, loss, count)
                    (c1._1 += c2._1, c1._2 + c2._2, c1._3 + c2._3)
                })

          if (miniBatchSize >; 0) {
              stochasticLossHistory.append(lossSum / miniBatchSize + regVal)
              val update = updater.compute(
                  weights, Vectors.fromBreeze(gradientSum / miniBatchSize.toDouble),
                  stepSize, i, regParam)
              weights = update._1
              regVal = update._2
          } else {
              logWarning(s"Iteration ($i/$numIterations). The size of sampled batch
                 is zero")
          }
      }

    logInfo("GradientDescent.runMiniBatchSGD finished. Last 10 stochastic losses
        %s".format(stochasticLossHistory.takeRight(10).mkString(", ")))

    (weights, stochasticLossHistory.toArray)

    }
}


object LBFGS extends Logging {
    /**
     * LBFGS算法
     */
    def runLBFGS(
        data: RDD[(Double, Vector)],
        gradient: Gradient,
        updater: Updater,
        numCorrections: Int,
        convergenceTol: Double,
        maxNumIterations: Int,
        regParam: Double,
```

```
        initialWeights: Vector): (Vector, Array[Double]) = {

      val lossHistory = new ArrayBuffer[Double](maxNumIterations)

      val numExamples = data.count()

      val costFun =
          new CostFun(data, gradient, updater, regParam, numExamples)
      // 基于Breeze库的实现
      val lbfgs = new BreezeLBFGS[BDV[Double]](maxNumIterations, numCorrections,
          convergenceTol)

      val states =
          lbfgs.iterations(new CachedDiffFunction(costFun),
          initialWeights.toBreeze.toDenseVector)

      var state = states.next()
      while(states.hasNext) {
          lossHistory.append(state.value)
          state = states.next()
      }
      lossHistory.append(state.value)
      val weights = Vectors.fromBreeze(state.x)

      logInfo("LBFGS.runLBFGS finished. Last 10 losses %s".format(
          lossHistory.takeRight(10).mkString(", ")))

      (weights, lossHistory.toArray)
    }
  }
```

可以看到，`LogisticRegressionWithSGD`的实现关键是利用`broadcast`函数来广播每一轮的新权重（weight），利用`treeAggregate`函数来收集每一份梯度和损失（loss）；`LogisticRegressionWithLBFGS`中`CostFun`的实现，也是主要通过用`broadcast`函数和`treeAggregate`来工作。可以看到，两个函数都会输出最近10次迭代的`loss`值，这个有利于我们判断是否收敛。

8.3.4　模型评估

出于模型调优、效果监控等目的，我们需要对模型训练好的模型进行评估。在点击率预估这个场景中，我们通常从这几个指标进行评估。

1. logLoss

`logloss`实际上就是8.3.3节提到的对数似然损失函数的值。对数似然损失衡量了点击事件在当前观测数据（搜索日志）中发生的可能性：对数似然越小，发生的可能性越大，模型的预估性能越好。由于广告的点击率低，样本负例远远多于正例，`logloss`的值一般较小。在离线评估过程中，我们常常比较变化的比例，而不是绝对值。

在本章案例中我们通过上一节代码中的`logInfo`输出最近10次损失作为参考。

2. AUC

信息检索（IR）领域中常用的是召回率（recall）和准确率（precision）。但有时候，这两个指标不太好用。

举个例子，测试集合中有A类样本90个，B类样本10个。分类器C1把所有的测试样本都分成了A类，分类器C2把A类的90个样本分对了70个，B类的10个样本分对了5个。那么C1的准确率为90%，C2的准确率为75%。但是，显然C2更有用些。正负例分布不平衡时，AUC（Area Under roc Curve）这时候就更有用了。顾名思义，AUC的值就是处于ROC曲线下方的那部分面积的大小。先来了解下ROC（Receiver Operating Characteristic），ROC指标主要分析的是一个画在二维平面上的曲线——ROC曲线。二维平面的横坐标是伪阳率FPR（False Positive Rate），纵坐标是正阳率TPR（True Positive Rate）。对于某个分类器而言，我们在测试集合上的测试结果得到一个TPR和FPR点对，这样这个分类器就映射成ROC平面上的一个点。然后，我们调整这个分类器分类时使用的阈值为从0到1变化，就可以得到一个经过(0, 0)、(1, 1)的曲线，所以ROC曲线反应了正负例判定阈值变化时，正阳率（True positive rate）和伪阳率（False positive rate）的变化情况，参见图8-9。

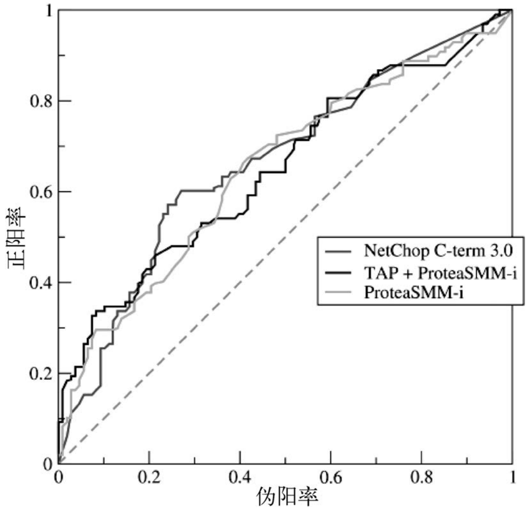


图8-9　ROC曲线示例（来自维基百科，另见彩插图8-9）

AUC最直接的含义是ROC曲线下的面积。使用随机分类器时AUC为0.5，当所有正例样本的预估值都大于所有负例样本时AUC的值为1，所以AUC值是越大越好。AUC反映了正负例样本预估值排序的正确性，同时不受正负例分布不平衡的影响。值得一提的是，当训练样本不充分，特征空间较小时，正负例样本无法相互区分，在这种情况下AUC的上界不可能为1。与`logloss`类似，我们通常比较不同模型变化的比例，而不是绝对值。在实践过程中我们发现，AUC提升幅度

大于0.3%时，效果一般都会有改善。AUC的计算有很多优化的版本，一个简单算法是对样本按预估值倒序排序后，遍历样本计算正阳率和伪阳率得到ROC点，同时累积两个相邻的ROC点间构成的梯形面积即可。

在本章的例子中，我们通过MLlib中的`BinaryClassificationMetrics`类来计算该指标。

3. MAE

Mean Absolute Error（平均绝对误差）也是模型评估的常用指标。点击率是一个统计值，通常需要对每个维度聚合以后来计算实际点击率，然后再计算MAE。为了简单起见，这个场景中我们定义如下：

$$MAE = \frac{1}{n}\sum_{i=1}^{n} | Predict_i - Lable_i | = \frac{1}{n}\sum_{i=1}^{n} | e_i |$$

这里Predict表示模型预测值，表示点击率，Lable表示实际是否点击（0 或者1）。

在本章的例子中，我们通过MLlib中的`RegressionMetrics`类来计算该指标。

8.3.5　数据准备

腾讯SOSO是腾讯公司自主研发的搜索引擎，于2009年9月上线。2012 KDD CUP比赛使用的数据就是腾讯SOSO提供的数据，具体参见：https://www.kddcup2012.org/c/kddcup2012-track2/data。

数据包括：搜索会话日志文件training.txt、查询词queryid_tokensid.txt、用户的属性userid_profile.txt。当然，出于对隐私的考虑，涉及用户和广告主的相关属性都做了匿名化处理。

针对training.txt，本章示例代码利用的几个重要字段如下。

1. `Click`：用户点击`AdID`的次数。
2. `Impression`：`AdID`在`QueryID`中被展示的次数。
4. `AdID`：展示的广告对应的广告ID。
5. `AdvertiserID`：广告ID对应的广告主ID。
8. `QueryID`：搜索引擎对query的编号。我们可以将这个编号作为索引，从数据文件'queryid_tokensid.txt'查找到这个query拆分以后对应的token列表。
12. `UserID`：用户编号。我们可以使用这个编号作为索引，从数据文件`'userid_profile.txt'`查找到每个`user_id`对应的性别和年龄信息。

queryid_tokensid.txt描述了每个`queryid`对应的`tokensid`。userid_profile.txt描述了每个用户ID对应的性别和年龄信息。其中，`'1'`表示男性，`'2'`表示女性，`'0'`表示未知。`'1'`表示(0, 12]岁，`'2'`表示(12, 18]岁，`'3'`表示(18, 24]岁，`'4'`表示(24, 30]岁，`'5'`表示(30, 40]岁，`'6'`表示40岁以上。这就是我们在前面说的离散值特征。

在上一节中，我们提到逻辑回归是一个线性模型，模型中某一维的特征不会和其他特征进行

组合作用，所以我们需要对特征进行交叉。比如，将广告ID（A）和用户性别（女）进行交叉形成一维特征：表示女性用户对于广告A这样一个维度的特征。当然这样的组合很多，实际组合以后的空间可能高达几亿维，然后我们还需要处理成在8.2.1节中提到稀疏向量的形式。为此，我们还需要对每种组合编一个号，作为稀疏向量的下标。

下面通过一个简单的例子来说明特征交叉。我们根据training.txt、queryid_tokensid.txt、userid_profile.txt的格式构造了3个测试文件，分别是：

```
root@m0:~# cat training_test.txt
0 1 1 1 91
0 2 1 2 92
1 1 2 1 93
格式是：点击  曝光  查询词ID（QueryID）  用户ID 其他属性

root@m0:~# cat queryid_tokensid.txt
1 234|445
2 444|55|675
格式是：queryid  token1 | token2

root@m0:~# cat userid_profiel_test.txt
1 1 3
2 2 5
格式是：userid  gender age
```

进行特征交叉组合以后，training_test.txt就对应变成以下形式：

```
0 1 16 1 21 1 17 1 19 1 18 1 20 1
0 2 2 1 5 1 3 1 6 1 4 1 1 1
1 1 7 1 14 1 10 1 13 1 11 1 15 1 9 1 12 1 8 1
格式是：  点击 曝光  特征编号  1  特征编号  1  特征编号  1  ...
```

第一行第三列的16表示第16维特征不为空，这里使用的特征都是8.1.4节中提到的二值特征。

除了对特征进行交叉、编号，我们还需要对训练样本进行异常值处理，比如点击大于曝光的样本、曝光为0的样本等，在后续的代码中你可以看到这些逻辑。

8.3.6 模型训练

现在，我们将实现上一节提到的特征交叉等逻辑代码，并处理成MLlib所需要的格式，进行模型训练，再结合8.3.4节提到的指标对模型进行评估。我们直接看LogisticRegression-Example.scala的源代码：

```
/*
LogisticRegressionExample.scala
author: linshifei  <linshifei365@gmail.com>
version: 2014.12.24

代码变量命名遵循作者的个人习惯
由于数据量比较大，根据读者的集群情况，代码运行可能需要修改的默认配置有：
```

```
      spark.local.dir
      spark.executor.memory
      spark.driver.maxResultSize
      spark.akka.frameSize
      spark.default.parallelism
      spark.network.timeout
      spark.akka.timeout
 */
import java.util.Date
import java.text.SimpleDateFormat
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
import org.apache.spark.mllib.classification.{LogisticRegressionWithSGD,
LogisticRegressionWithLBFGS,LogisticRegressionModel}
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.evaluation.{RegressionMetrics,
    BinaryClassificationMetrics}
import org.apache.spark.mllib.optimization.{SquaredL2Updater, L1Updater}
import org.apache.spark.rdd.RDD
import org.apache.spark.mllib.util.MLUtils

object LogisticRegressionExample {

    def main(args: Array[String]) {

        val conf = new SparkConf()
        val sc = new SparkContext(conf)

        var num_iterations = 50
        var train_algorithm = "SGD"
        var train_file_path = "hdfs:///data/track2_datasets_files_for_competitors/"
        var training_file_name = train_file_path + "training1kw.txt"
        var clear_threshold_test = false
        var reg_type = "L1"


        // 参数解析可以使用scopt.OptionParser
        if(args.length >= 1){
            train_file_path =  args(0)
        }

        if(args.length >= 2){
            training_file_name = train_file_path  + args(1)
        }

        if(args.length >= 3){
            num_iterations = args(2).toInt
        }

        if(args.length >= 4){
            train_algorithm = args(3)
        }

        if(args.length >= 5){
```

```
    if(args(4) == "L2"){
        reg_type = "L2"
    }
}

println("using train_file_path :" + train_file_path)
println("using training_data :" + training_file_name)
println("num_iterations :" + num_iterations.toString())
println("train_algorithm :" + train_algorithm)
println("reg_type :" + reg_type)

// 使用kddcup2012数据集，参见：http://www.kddcup2012.org/c/kddcup2012-track2
val training_data_raw = sc.textFile(training_file_name)
val qid = sc.textFile(train_file_path + "queryid_tokensid.txt")
val uid = sc.textFile(train_file_path + "userid_profile.txt")
val keywords = sc.textFile(train_file_path +
    "purchasedkeywordid_tokensid.txt")
val title = sc.textFile(train_file_path + "titleid_tokensid.txt")
val description = sc.textFile(train_file_path + "descriptionid_tokensid.txt")

// keywords格式是：id x|y
// 转换为id为key，其他为value (array)
val m_keywords = keywords.map(line => {
    val arr=line.split('\t');
    (arr(0), arr(1).split('|'))
}).reduceByKey((a1, a2) => a1).cache()

// title格式是：id x|y
// 转换为id为key，其他为value (array)
val m_title = title.map(line => {
    val arr=line.split('\t');
    (arr(0), arr(1).split('|'))
}).reduceByKey((a1, a2) => a1).cache()

// description格式是：id x|y
// 转换为id为key，其他为value (array)
val m_description = description.map(line => {
    val arr=line.split('\t');
    (arr(0), arr(1).split('|'))
}).reduceByKey((a1, a2) => a1).cache()

// uid格式是：uid 1 2 3
// 转换为uid为key，其他为value (array)
val m_uid = uid.map(line => {
    val uid_array=line.split('\t');
    (uid_array(0), uid_array.drop(1))
}).reduceByKey((a1, a2) => a1)


// qid格式是：qid x|y
// 转换为qid为key，其他为value (array)
val m_qid = qid.map(line => {
    val arr=line.split('\t');
    (arr(0), arr(1).split('|'))
}).reduceByKey((a1, a2) => a1)
```

```
// 给每个session加一个编号
val training_data_raw_idx = training_data_raw.zipWithIndex().map(x =>
    (String.valueOf(x._2), x._1))

/*
展开training_data_raw，并保留原行数据 ：
1. Click：点击数
2. Impression：展现数
4. AdID：广告ID
5. AdvertiserID：广告主ID
8. QueryID：查询词ID
9. KeywordID：关键词ID
10. TitleID：广告标题ID
11. DescriptionID：广告描述ID
12. UserID：用户ID
主要过程是：line => (("type",i,p._2(1)), instance)
其中，instance(7)是query id，instance(11)是user id……
join默认两边都有的记录
*/
val train_instance_expand = training_data_raw_idx.map(l => {
    val arr = l._2.split('\t')
    // instance只保留：index、Click、Impression、UserID、AdID、AdvertiserID、
    // query、title、description、Position
    (arr(8), Array(l._1,arr(0),arr(1),arr(11),arr(3),arr(4),arr(7),arr(9),
        arr(10),arr(6)))
}).join(m_keywords).map(a => (a._2._1(7),a._2)  // value :  instance、
                                                 // keywords_token
).join(m_title).map(a => (a._2._1._1(8),(a._2._1._1,a._2._1._2,a._2._2))
                                                 // value :  instance、
                                                 // keywords_token、
                                                 // title_token
).join(m_description).map(a =>
    (a._2._1._1(6),(a._2._1._1,a._2._1._2,a._2._1._3,a._2._2))
                                                 // value :  instance、
                          // keywords_token、title_token、description_token
).join(m_qid).map(a => (a._2._1._1(3),
    (a._2._1._1,a._2._1._2,a._2._1._3,a._2._1._4,a._2._2))
// value : instance、keywords_token、title_token、description_token、qid_arr
).join(m_uid).map(a =>
    (a._2._1._1,a._2._1._2,a._2._1._3,a._2._1._4,a._2._1._5,a._2._2)
// value : instance、keywords_token、title_token、description_token、qid_arr、
      uid_arr
).flatMap( p => {  // p的格式是1:instance、2:keywords_token、3:title_token、
                   // 4:description_token、5:qid_arr、6:uid_arr

var array_size = p._6.length             // AdvertiserID和user交叉
array_size =  array_size + p._5.length  // AdID和query_token交叉，
                                         // 每次请求AdID只有一个
array_size =  array_size + p._5.length  // AdvertiserID和query_token交叉
array_size =  array_size + 3  // title、keyword、description是否包含query

val a = new Array[((String,String, String), Array[String])](array_size)
var n = 0
```

```
    var token_hit_count = 0

    token_hit_count = checkTokenMatch(p._2,p._5)
    a(n) = (("1","1",token_hit_count.toString()), p._1) // 标识第一类特征 : title、
                                        // keyword、description是否包含query
    n += 1

    token_hit_count = checkTokenMatch(p._3,p._5)
    a(n) = (("1","2",token_hit_count.toString()), p._1)
    n += 1

    token_hit_count = checkTokenMatch(p._4,p._5)
    a(n) = (("1","3",token_hit_count.toString()), p._1)
    n += 1

    for (i <- p._5){                    // 展开query_token
        a(n) = (("2",i,p._1(4)), p._1) // 标识第二类AdID和query_token交叉,
                                       // p._1是instance
        n += 1
    }

    for (i <- p._6){                      // 展开user
        a(n) = (("3",i,p._1(5)), p._1) // 标识AdvertiserID和user交叉
        n += 1
    }

    for (i <- p._5){                     // 展开query_token
        a(n) = (("4",i,p._1(5)), p._1) // 标识AdvertiserID和query_token交叉
        n += 1
    }

    a
}).cache()

//train_instance_expand是(feature_list,instance),
//比如：(("type",i,p._2(1)),p._1) 需要去重,
//本地加编号n
var n = 0

val feature_index = train_instance_expand.reduceByKey((a1, a2) =>
    a1).map(_._1).collect().map(a => {
    n += 1
    (a, n)
})

// collect以后是数组arrray，重新回到RDD
val feature_index_rdd = sc.parallelize(feature_index)

//以每个instance的feature列表作为key进行join，join得到每一种instance需要的feature
//编号：((feature_list) ,(instance,feature_id))
//然后反转：以每一个instance为key，这里需要倒过来，即对同一个组合groupByKey
val train_instance = train_instance_expand.join(feature_index_rdd).map(a =>
    {(a._2._1.mkString("\t"), a._2._2)}
```

8

```
  ).groupByKey().map( a => {
      val arr = a._1.split('\t')  // mkstring以后是字符串
      var s = ""
          for(n <- a._2) {    // groupByKey以后，同一个类型的特征组合
                              // 所有的编号在a._2中
              s = s + "\t" + n + "\t1"  // 这里使用稀疏矩阵，1表示这一维不为空
          }
      arr(1) + "\t" + arr(2) + s
  })

// 这里需要过滤“点击大于曝光”和“ 曝光大于0”的两种情况
// zipWithIndex的作用是将每个元素和其所在的下标组成一个pair
val train_instance_filter =
    train_instance.map(_.split("\t").zipWithIndex).filter(arrWithIdx =>
    arrWithIdx(0)._1.toInt <= arrWithIdx(1)._1.toInt && arrWithIdx(1)._1.toInt
    > 0
)

// 最大的编号
val maxSize = n + 1

val train_instance_flat = train_instance_filter.flatMap { arrWithIdx => {
    val feature_index_arr = arrWithIdx.filter(i => i._2 > 1 && i._2 % 2 ==
        0).map(_._1.toInt)
    val feature_value_arr = arrWithIdx.filter(i => i._2 > 1 && i._2 % 2 ==
        1).map(_._1.toDouble)

    // 点击数，正例数
    val click_count = arrWithIdx(0)._1.toInt
    // 曝光数，负例数
    val imp_count = arrWithIdx(1)._1.toInt
    var outs = new Array[(LabeledPoint)](imp_count)

    for (i <- click_count until imp_count)
        outs(i-click_count) = LabeledPoint(0, Vectors.sparse(maxSize,
            feature_index_arr, feature_value_arr))

    for (i <- 0 until click_count )
        outs(i+imp_count-click_count) = LabeledPoint(1,
            Vectors.sparse(maxSize, feature_index_arr, feature_value_arr))
    outs
}}

val examples = train_instance_flat.randomSplit(Array(0.99, 0.01), seed = 11L)
val training_set = examples(0).cache()
val test_set = examples(1).cache()

val num_training = training_set.count()
val num_test = test_set.count()
println(s"Training: $num_training, Testing: $num_test.")

val updater = reg_type match {
    case "L1" => new L1Updater()
    case "L2" => new SquaredL2Updater()
}
```

```
    val model = train_algorithm match {
        case "LBFGS" =>
            val algorithm = new LogisticRegressionWithLBFGS()
            algorithm.optimizer
                .setNumIterations(num_iterations)
                .setUpdater(updater)
            algorithm.run(training_set)
        case "SGD" =>
            val algorithm = new LogisticRegressionWithSGD()
            algorithm.optimizer
                .setNumIterations(num_iterations)
                .setUpdater(updater)
            algorithm.run(training_set)
    }

    model.clearThreshold()

    val weight_count = model.weights.size
    println(s"\nModel weights count :${weight_count}")

    val prediction = model.predict(test_set.map(_.features))
    val score_and_labels = prediction.zip(test_set.map(_.label))
    val metrics = new BinaryClassificationMetrics(score_and_labels)
    println(s"\nTest areaUnderROC = ${metrics.areaUnderROC()}.")

    val metrics2 = new RegressionMetrics(score_and_labels)
    println(s"\nTest  meanAbsoluteError = ${metrics2.meanAbsoluteError}")

    val  date_format = new SimpleDateFormat("yyyyMMddHHmm")
    val model_path =  train_file_path + "model_"  + date_format.format(new Date())
    model.save(sc, model_path)
    sc.stop()

    }

    def checkTokenMatch(token1: Array[String],token2: Array[String]) : Int = {
        if(token1.length ==0 || token2.length ==0){
            return 0
        }

        var hit = 0
        for( item1 <- token1){
            for(item2 <- token2){
                if(item1 == item2){
                    hit += 1
                }
            }
        }
        return hit
    }
}
```

注意，这里的RDD调用`cache()`的代码，应该根据自己的集群内存情况来调整，否则会出错，

内存比较小的可以使用persist(StorageLevel.MEMORY_AND_DISK)来替换。当训练数据集合比较大，资源有限的时候，我们也可以对训练集合进行一定比例的采样。

示例用到的代码目录如下：

```
shifei@shifei-vm:~/project/spark_book/LogisticRegressionExample$ ls
project  simple.sbt  src  target

shifei@shifei-vm:~/project/spark_book/LogisticRegressionExample/src/main/scala$ ls
LogisticRegressionExample.scala
```

这里使用sbt构建jar包，sbt的用法可以参考前面的章节。

```
shifei@shifei-vm:~/project/spark_book/LogisticRegressionExample$ cat simple.sbt
name := "LogisticRegressionExampleProject"
version := "1.8"
scalaVersion := "2.10.4"
val revision = "1.4.0" // Spark版本号
libraryDependencies += "org.apache.spark" %% "spark-core" % revision
libraryDependencies += "org.apache.spark" % "spark-mllib_2.10" % revision
```

输入sbt package开始打包构建，sbt会根据simple.sbt描述的版本下载所需要的库，第一次会比较慢。

```
shifei@shifei-vm:~/project/spark_book/LogisticRegressionExample$ sbt package
[info] Set current project to LogisticRegressionExampleProject (in build
file:/home/shifei/project/spark_book/LogisticRegressionExample/)
[info] Compiling 1 Scala source to
/home/shifei/project/spark_book/LogisticRegressionExample/target/scala-2.10/
classes...
[info] Packaging
/home/shifei/project/spark_book/LogisticRegressionExample/target/scala-2.10/
logisticregressionexampleproject_2.10-1.8.jar ...
[info] Done packaging.
[success] Total time: 20 s, completed Dec 25, 2015 10:55:28 PM
```

然后通过以下命令提交任务：

```
/usr/local/myhadoop/spark/bin/spark-submit \
    --deploy-mode cluster \
    --driver-cores 8 \
    --driver-memory 20G \
    --total-executor-cores 40 \
    --class LogisticRegressionExample \
    hdfs:///data/logisticregressionexampleproject_2.10-1.8.jar training1kw.txt  100
LBFGS
```

注意，代码中提到集群的参数一定要提前确认是否已经修改，没有修改也可以在提交命令中指定，比如：--conf "spark.kryoserializer.buffer.max=1g" 。

提交以后可以通过Spark提供的UI界面查看任务进度，界面还会提供整个集群的资源信息，比如：

```
·          **URL:** spark://m1.hadoop:7077
·          **REST URL:** spark://m1.hadoop:6066 _(cluster mode)_
·          **Workers:** 5
·          **Cores:** 40 Total, 33 Used
·          **Memory:** 142.5 GB Total, 120.0 GB Used
·          **Applications:** 1 Running, 6 Completed
·          **Drivers:** 1 Running, 7 Completed
·          **Status:** ALIVE
```

表8-4所示是一个完成训练的示例。

表8-4 Spark任务UI示例

应用ID（Application ID）	核数（Core）	每个节点内存（Memory per Node）	用户名（User）	状态（State）	运行时长（Duration）
app-20151202193002-0003	32	24.0 GB	hadoop	FINISHED	1.5 分钟

注意，训练的模型在实际使用的时候还需要注意几个问题：

- 预估使用的特征编号要对齐训练这里的特征编号；
- 如果训练的模型用于排序过程的点击率预估，需要调用`model.clearThreshold()`，保证模型输出是小数值；
- 模型校验、模型传输等过程可能出错。

8.3.7 模型调优

除了数据准备阶段比较费时，模型调优也是比较耗时的一个环节。调优主要集中于特征选择和训练参数选择，实际应用中往往要在效率和效果中间寻找平衡。

本章的示例代码，在腾讯企业云的“攻城之地”活动所提供的机器集群上，做了几组参数配置的模型对比，参见表8-5。

表8-5 模型对比

模型	算法	样本量	特征类型	特征维度	最大迭代	LOSS	MAE	AUC
1	SGD	1000 w	6	9 887 394	100	0.2024	0.1343	0.5911
2	LBFGS	1000 w	6	9 887 394	100	0.1551	0.0720	0.7606
3	LBFGS	1000 w	7	9 887 397	100	0.1597	0.0737	0.7634
4	LBFGS	2000 w	6	15 985 071	100	0.1668	0.0746	0.7571
5	LBFGS	2000 w	7	15 985 074	100	0.1652	0.0738	0.7644

可以看到，LOSS、MAE和AUC在模型效果衡量上基本是相关的：AUC越高，LOSS和MAE越小，比如：对比模型1和模型2。其中，SGD算法收敛很慢，而LBFGS算法基本在50次迭代左右就接近收敛。SGD算法在100次迭代以后LOSS还是很大，尝试把SGD最大迭代次数调到2000，耗时100多分钟，算法还是没有收敛，基本上不可用。我们也可以调整两次迭代损失函数变化的容

忍度（convergenceTol）来改变训练时间。convergenceTol默认是0.0001，从日志看，都没有达到convergenceTol的阈值，一般是最大迭代次数在控制训练时间，LBFGS算法的最大迭代次数设置为100，迭代最后几次的LOSS减小就很小了，所以这里都设置100进行效果对比 。

当数据集变化时，AUC指标也会变化。这里的简单做法是，在同一个数据集上取多少次计算的平均值。很明显可以看出，增加有效的特征以后，AUC指标有所提升，模型3对比模型2有0.37%的提升，模型5对比模型4有0.96%的提升，这里添加的特征是展现位置等特征。事实上，kddcup2012数据集的数据还可以添加的特征有很多，这里的实验只使用了7类特征。值得注意的是，不同场景需要的特征作用不太一样。在搜索广告案例中可以发现，用户特征（数据集中`UserID`）对效果提升不大，而展现位置信息（数据集中`Position`）对效果提升明显。而在展示广告场景中，用户特征作用可能更大。

此外，模型4对比模型2，训练数据增加以后，特征数量增加明显，从9 887 394到15 985 071。这里特征数量增加主要是广告ID和广告主ID的交叉类特征，模型在特征表达上更精准了，覆盖了更多场景，范化能力更好。由于2000 w和1000 w对应使用的数据集合有较大变化，这里的AUC指标不太可比。点击率预估场景中，我们通常保留所有的正样本，对负样本进行采样，同时也会对时间上相对比较旧的样本做降权处理。

MLlib中LBFGS算法实现上的正则项（regularization parameter）默认是0.0，为了防止过拟合可以通过`setRegParam`接口进行修改，一般设置成`0.001`。

最后模型是否上线，通常我们还是通过不同模型中线上A/B测试的效果来进一步评估。

为了验证测试数据集合大小和训练时间的关系，我们分别测试了100 w 、1000 w、2000 w这些不同大小的训练集合，迭代50次的训练时间变化。从表8-6可以看出，训练时间基本上随着训练数据集合大小的变化线性增加。在Spark平台上，我们可以很方便地通过增加硬件资源来缩短训练时间。但受限于算法每一轮迭代的广播新权重和收集梯度的两个过程，driver节点可能是一个瓶颈，具体情况要综合看driver节点机器负载来调整集群硬件配置。

表8-6　不同大小数据集合的耗时对比

应用ID（Application ID）	核数（Core）	每个节点内存（Memory per Node）	用户名（User）	状态（State）	运行时长（Duration）
app-20151202193002-0003	32	24.0 GB	hadoop	FINISHED	1.5 min
app-20151202165133-0002	32	24.0 GB	hadoop	FINISHED	23 min
app-20151202154403-0001	32	24.0 GB	hadoop	FINISHED	58 min

分析不同Job的耗时，我们发现时间主要集中耗在`treeAggregate`和`broadcast`等过程，这点符合预期。在实际应用过程中，我们还可以结合Pipeline API完成更多工作，比如模型交叉验证等。

事实上，要做好广告点击率预告，还有很多的工作要做，也需要不小的人力投入，这里只是做了一个简单的案例，供新手学习。同时，从这个案例也可以看到，MLlib的LBFGS算法在千万维级别特征的LR模型训练上，已经达到工业应用的初步要求。

附　录

Scala语言参考

为了方便那些没有使用过Scala语言的读者，本章带大家快速了解一下。有基本编程经验的读者在学习完本章后，就可以用Scala编写简单的Spark程序了。

Scala 语言基础

本节介绍Scala语言的基础知识，包括安装、变量命名、表达式、基本类型、控制结构等。

关于 Scala

Scala是“Scalable Language”的缩写，是一门可伸缩的语言，即可以当脚本使用，又可以构造大型系统（比如Spark）。Scala非常完美地整合了面向对象编程和函数式编程两种特性。它即支持纯面向对象编程，每一个值都是对象，每一次运算都是一次方法调用，而且同时支持函数式编程，所有的函数都是对象。Scala是一种静态语言，通过编译时的代码检查来保证代码的安全性和一致性，同时也可以像动态语言那样支持交互式编程。Scala运行于JVM之上，所有Scala代码最终都会转换成Java字节码来运行。并且，Scala与Java无缝集成，在Scala中可以像调用Scala库一样来调用Java库。

Scala 安装

第一步，从Java官方网站下载JDK并安装（支持1.6及以上版本），并设置好`JAVA_HOME`环境变量。

第二步，从Scala官方网站http://www.scala-lang.org/download/下载最新的Scala语言安装包，解压缩后可以直接使用。可执行文件在bin目录下，Scala是交互式解释器，`scalac`是编译器（注：windows下加.bat后缀）。此外，官网上也有IDE可供下载。

第三步，进入Scala交互式编程环境，Linux下可以执行./bin/scala。

```
$ scala
Type :help for more information.
```

```
scala>
```

基础知识

❑ Hello world程序

在Scala交互式编程环境下，直接输入Scala代码：

```
scala> println("Hello, world!")
Hello, world!
```

❑ 表达式

```
scala> 1 + 1
res0: Int = 2
```

`res0`是解释器自动生成的变量名，用来代表表达式的计算结果。它是`Int`类型，值为`2`。

注意　Scala中几乎一切都是表达式。

❑ 变量

Scala有两种变量：`val`、`var`。

val表示常量，一旦定义就不能重新赋值，比如上面的`1+1`等价于：

```
scala> val two = 1 + 1
two: Int = 2
```

`two`被定义为`val`类型，不能再被赋值，下面的语句会报错：

```
scala> two = 2
<console>:8: error: reassignment to val
    two = 2
   ^
```

`var`表示变量，定义之后还可以重新赋值。对于上面的例子，如果想重新赋值，可以用`var`：

```
scala> var two = 1 + 1
two: Int = 2
```

此时可以重新赋值：

```
scala> two = 2
two: Int = 2
```

❑ 标识符

标识符用于变量名、类名等。Scala有4种标识符，如下。

- **字母数字式**。大多数语言使用的形式，以字母或下划线开头，后面接字母、数字或下划线。一般遵循驼峰命名法，比如`toString`、`HashSet`，而非`tostring`。
- **运算符式**。非字母、数字的可打印ASCII字符都算，比如数学运算符`+-*/%<>`，一般用于方法定义。
- **混合式**。格式为“字母数字式标识符_运算符式标识符”，比如`unary+`，`myvar=`。
- **文字式**。Scala支持使用任意字符（比如关键词）作为标识符，用`包起来，里面的任何字符都会保留原样，比如`` `x` ``、`<clinit>`、`` `yield` ``。

注意　标识符区分大小写。

❑ 面向对象

Scala是面向对象的，每一个值都是对象，每一次运算都是一次方法调用。

比如表达式`1+2`，其中`1`和`2`都是`Int`对象，`+`运算符对应的是`Int`的`+`方法。所有`1+2`都相当于`Int`类型对象`1`调用`+`方法，参数是`Int`类型`2`。

所以下面两个表达式是等价的：

```
scala> val three = 1 + 2
three: Int = 3

scala> val three = (1).+(2)
three: Int = 3
```

❑ 函数

我们可以用`def`创建函数：函数体超过1行时，可以用`{}`包裹起来。

```
scala> def addOne(x: Int): Int = x + 1
addOne: (x: Int)Int
```

函数`addOne`接受1个`Int`类型的输入参数（类型必须显式定义），返回`Int`类型。

调用函数：

```
scala> addOne(1)
res1: Int = 2
```

函数体超过 1 行时，可以用`{}`包裹起来。

注意　Scala支持函数式编程，且函数像其他值一样是“一等公民”。

❑ 匿名函数

我们可以创建匿名函数：

```
scala> (x: Int) => x + 1
res2: Int => Int = <function1>
```

并使用系统自动生成的函数名来调用它：

```
scala> res2(1)
res3: Int = 2
```

匿名函数可以赋值给一个值，因为Scala是面向对象的，一切都是对象；函数也是：

```
scala> val addOne = (x: Int) => x + 1
addOne: Int => Int = <function1>

scala> addOne(1)
res4: Int = 2
```

Scala中，表达式结尾处的`;`不是必需的，你只要确保Scala可以准确区分出不同的表达式并且不产生歧义即可。

基本类型

Scala的基本类型有`Byte`、`Short`、`Int`、`Long`、`Char`、`String`、`Float`、`Double`、`Boolean`。

Scala支持隐式类型转换，所以虽然是静态语言，但使用体验上接近动态语言。比如，基本类型大部分不用显式定义就可以直接使用。

数字一般为`Int`类型，以`0x`开头表示十六进制，以`L`结尾表示`Long`。

```
scala> 3
res0: Int = 3

scala> 0x0F
res1: Int = 15

scala> 3L
res2: Long = 3
```

`Byte`、`Short`类型需要显式指定：

```
scala> val little: Byte = 3
little: Byte = 3

scala> val little: Short = 3
little: Short = 3
```

`Char`和`String`支持`\`转义：

```
scala> val a = 'a'
a: Char = a

scala> val str = "Hello \101"
str: String = Hello A
```

带小数点形式的默认为Double类型，以F结尾表示Float：

```
scala> val pi = 3.14F
pi: Float = 3.14

scala> val pi = 3.14
pi: Double = 3.14
```

控制结构

Scala内置的控制结构有if、while、for、try、match，而且Scala的可伸缩性还支持通过库的形式扩展控制结构。

另外，每一个控制结构也都是表达式，是有值的。

- **if表达式**

```
val filename =
if (!args.isEmpty) args(0)
else "default.txt"
```

- **while循环表达式**

```
// 计算最大公约数
def gcdLoop(x: Long, y: Long): Long = {
    var a = x
    var b = y
    while (a != 0) {
        val temp = a
        a = b % a
        b = temp
    }
    b
}
```

- **do...while**

```
// 遍历读文件
var line = ""
do {
    line = readLine()
    println("Read: "+ line)
} while (line != "")
```

- **for表达式**

```
scala> for (i >1 to 4)
println("Iteration "+ i)
```

- **try...catch...finally**

```
scala> def g(): Int = try{1/0}catch{case ex: ArithmeticException => 2} finally
```

```
{println("finally")}
g: ()Int

scala> g()
finally
res6: Int = 2
```

❑ actor“控制结构”

actor是基于消息传递的并发计算模型，Scala也支持，使用起来不像方法调用，倒是像某种控制结构。其实它不是Scala内置的，而是通过库来实现的，这也是Scala可伸缩性的典型代表。

```
actor {
    var sum = 0
    loop {
        receive {
            case Data(bytes) => sum += hash(bytes)
            case GetSum(requester) => requester ! sum
        }
    }
}
```

基本数据结构

Scala中常用的数据结构有数组、列表、元组、`Set`、`Map`等。

1. 数组`Array`

成员可修改，下标从`0`开始，并通过`()`（而不是`[]`）来引用：

```
val greetStrings = new Array[String](3)
greetStrings(0) = "Hello"
greetStrings(1) = ", "
greetStrings(2) = "world!\n"
```

2. 列表`List`

成员只读：

```
scala> val numbers = List(1, 2, 3, 4)
numbers: List[Int] = List(1, 2, 3, 4)

scala> numbers(0)
res3: Int = 1
```

3. 元组`Tuple`

成员只读，与`List`非常类似，但成员可以是不同类型：

```
scala> val hostPort = ("localhost", 80)
hostPort: (String, Int) = (localhost, 80)
```

不能用()方式来访问成员，而是“_下标”，下标从1开始：

```
scala> hostPort._1
res5: String = localhost

scala> hostPort._2
res6: Int = 80
```

4. 集合Set

没有重复的成员：

```
scala> Set(1, 1, 2)
res0: scala.collection.immutable.Set[Int] = Set(1, 2)
```

5. 映射Map

```
scala> val m = Map(1 -> "one", 2 -> "two")
m: scala.collection.immutable.Map[Int,String] = Map(1 -> one, 2 -> two)

scala> m(2)
res9: String = two
```

注意，这里的m是immutable只读。若要创建读写版本，需要显式指定类型：

```
scala> val m = scala.collection.mutable.Map[Int,String](1 -> "one", 2 -> "two")
m: scala.collection.mutable.Map[Int,String] = Map(2 -> two, 1 -> one)
```

添加元素：

```
scala> m += (3 -> "three")
res5: m.type = Map(2 -> two, 1 -> one, 3 -> three)
```

或者直接给键赋值：

```
scala> m(3) = "three"
```

删除成员：

```
scala> m -= 2
res11: m.type = Map(1 -> one, 3 -> three)
```

面向对象

Scala语言支持完整的面向对象编程，类似C++或Java，包括类继承、多态等。

定义类 class

类用关键字class来定义：

```
class Hello(_a: Int) {
```

```
    private val a = _a;
    println("initializing...(" + _a + ")");
    def Add(b: Int): Int = a + b;
    def this() = this(0);
}
```

默认构造函数不需要单独定义，直接放在class的定义中。其中，private表示私有。我们也可以添加其他构造函数，定义都是def this，只是参数不一样。

下面用new来创建类的实例：

```
scala> val a = new Hello(1)
initializing...(1)
a: Hello = Hello@f11a7de
```

继承

类继承用关键字extends表示，下面的类Hello2继承自Hello：

```
class Hello2(a: Int) extends Hello(a) {
    def Add(b: Int, c: Int): Int = a + b + c;
}
```

继承的同时重载了Add方法：

```
scala> val h2 = new Hello2(1)
initializing...(1)
h2: Hello2 = Hello2@2645d22d

scala> h2.Add(1)
res21: Int = 2

scala> h2.Add(1,2)
res22: Int = 4
```

单实例对象

Scala不支持static变量、成员，但提供了单例对象，用object来定义，类似于OO的单实例模式：

```
object Timer {
    var count = 0
    def currentCount(): Long = {
        count += 1
        count
    }
}
```

此时不需要new，直接使用：

```
scala> Timer.currentCount()
res0: Long = 1
```

单例对象可以和类具有相同的名字，放在同一个源文件中的话，它们可以互相访问彼此的`private`变量、方法：

```
class Bar(foo: String)
object Bar {
    def apply(foo: String) = new Bar(foo)
}
```

函数式

Scala语言最大的特色是函数式编程，能让程序非常简洁易读。

匿名函数

匿名函数是函数式编程最重要的特色之一。

```
scala> (x: Int) => x + 1
res0: Int => Int = <function1>

scala> res0(10)
res1: Int = 11
```

函数在Scala中是一等公民，可以赋值给变量：

```
scala> var increase = (x: Int) => x + 1
increase: (Int) => Int = <function>

scala> increase(10)
res2: Int = 11
```

匿名函数作为参数

这是匿名函数最常见的使用形式：

```
scala> val someNumbers = List(1, -3, -5, 9, 0)
someNumbers: List[Int] = List(1, -3, -5, 9, 0)

scala> someNumbers.filter((x) => x > 0)
res4: List[Int] = List(1, 9)
```

匿名函数的定义可以再简洁一些，比如去掉括号：

```
scala> someNumbers.filter(x => x > 0)
res4: List[Int] = List(1, 9)
```

如果匿名函数的参数只被使用一次，还可以更简洁，直接写函数体，用_代表一个参数：

```
scala> someNumbers.filter(_ > 0)
res5: List[Int] = List(1, 9)
```

函数嵌套

因为函数也是对象，可以像函数变量一样嵌套定义：

```
import scala.io.Source
def processFile(filename: String) {
    def processLine(line: String) {
        println(line)
    }
    Source.fromFile(filename).getLines.foreach(processLine)
}

processFile("t.scala")
```

部分应用函数

调用函数的过程，不需要一次准备好全部参数，而是可以分多步提供，来看下面的例子。

下面是一个普通求和的函数，需要两个输入参数：

```
scala> def adder(m: Int, n: Int) = m + n
adder: (m: Int,n: Int)Int
```

add2是定义的部分应用函数（partially applied function），只输入一个参数2，使用_代表暂时没有准备好的另外一个参数：

```
scala> val add2 = adder(2, _:Int)
add2: (Int) => Int = <function1>
```

调用部分应用函数，输入剩下的那个参数，完成整个adder调用：

```
scala> add2(3)
res1: Int = 5
```

柯里化函数

柯里化（currying，又译为“卡瑞化”或“加里化”）是把接受多个参数的函数变换成接受单个参数（最初函数的第一个参数）的函数，并且返回接受余下的参数而且返回结果的新函数的技术。

比如，在没有使用柯里化时，加法函数可能是像下面这样定义的，调用plainOldSum时，需要一次准备好两个参数：

```
scala> def plainOldSum(x:Int,y:Int) = x + y
plainOldSum: (x: Int, y: Int)Int

scala> plainOldSum(1,2)
res0: Int = 3
```

而使用柯里化之后，不需要一次性准备好两个参数，而是可以分两次传入。这是柯里化之后的加法定义：

```
scala> def curriedSum(x:Int)(y:Int) = x + y
curriedSum: (x: Int)(y: Int)Int
```

可以一次输入两个参数来调用：

```
scala> curriedSum (1)(2)
res0: Int = 3
```

但其实这里是有两次调用，`curriedSum (1)`调用返回一个新的函数，然后给这个新的函数传入参数2并调用。因此我们可以显式依次调用（注意，定义`onePlus`时使用了部分应用函数符号_）：

```
scala> val onePlus = curriedSum(1)_
onePlus: Int => Int = <function1>

scala> onePlus(2)
res1: Int = 3
```

通过柯里化，你还可以定义多个类似于`onePlus`的函数，比如`twoPlus`：

```
scala> val twoPlus = curriedSum(2) _
twoPlus: Int => Int = <function1>

scala> twoPlus(3)
res2: Int = 5
```

集合成员遍历

Scala提供了一些功能函数以方便对集合对象（比如`Array`、`List`、`Tuple`、`Set`、`Map`）进行遍历操作。功能函数的输入一般是函数，可能没有返回值，如果有的话一般是同类型值的新集合。

以`List`为例（其他集合与此基本相同）：

```
scala> val numbers = List(1, 2, 3, 4)
numbers: List[Int] = List(1, 2, 3, 4)
```

- **map**

`map`对列表中的每个元素应用一个函数，返回应用后的元素所组成的列表。

```
scala> numbers.map((i: Int) => i * 2)
res0: List[Int] = List(2, 4, 6, 8)
```

- **foreach**

foreach与map很像，但没有返回值。foreach仅用于有副作用（side-effects）的函数。

```
scala> numbers.foreach((i: Int) => i * 2)
```

- **filter**

filter用于过滤成员，返回新的集合，其中只保留计算结果为true的元素：

```
scala> numbers.filter((i: Int) => i % 2 == 0)
res0: List[Int] = List(2, 4)
```

高级

本节介绍一些高级功能，但这些功能也是Scala编程时经常使用的。

包

包用于管理各种名称，与Java中的包概念非常类似，但更灵活。

在文件头部用package定义包：

```
package org.scala.example

object Color{
    val BLUE = "Blue"
    val RED = "Red"
}
```

然后可以通过包名直接访问名称：

```
println("the color is: " + org.scala.example.Color.BLUE)
```

或者先import，再访问：

```
import org.scala.example.Color
println("the color is: " + Color.BLUE)
```

模式匹配

Scala语言的模式匹配功能类似于其他语言的switch...case，但功能更强大。

我们可以像使用switch...case一样来使用模式匹配：

```
val times = 1
```

```
times match {
    case 1 => "one"
    case 2 => "two"
    case _ => "some other number"
}
```

或者：

```
times match {
    case i if i == 1 => "one"
    case i if i == 2 => "two"
    case _ => "some other number"
}
```

更常见的使用方法是对象匹配：

```
def bigger(o: Any): Any = {
    o match {
        case i: Int if i > 0 => i - 1
        case i: Int => i + 1
        case d: Double if d > 0.0 => d - 0.1
        case d: Double => d + 0.1
        case text: String => text + "s"
  }
}
val v42 = 42
Some(3) match {
    case Some(`v42`) => println("42")
    case _ => println("Not 42")
}
```

Scala 作为脚本运行

Scala语言伸缩性很强，结合函数式编程代码简洁的特点，完全可以当作脚本语言来使用。

将以下内容保存到文件hello.scala：

```
#!/usr/bin/env scala

println("hello, " + args(0) + "!")
```

给hello.scala添加可执行权限：

```
> chmod +x hello.scala
```

然后像脚本一样执行：

```
> ./hello.scala scala

hello, scala!
```

Scala 应用程序

最后我们用Scala来写一个应用程序。

只有单实例对象才能作为Scala应用程序的入口，而且内部必须有一个`main`方法，`main`有且仅有一个`Array[String]`类型的参数，并且返回`Unit`。

将以下代码保存到文件Hello.scala：

```
object Hello {
    def main(args: Array[String]) {
        for ( arg <- args )
            println("Hello, " + arg)
    }
}
```

用`scalac`命令来编译源文件：

```
> scalac Hello.scala
```

可以看到这里生成了3个.class文件：

```
total 16
-rw-r--r-- 1 root root 1312 Jun  7 00:55 Hello$$anonfun$main$1.class
-rw-r--r-- 1 root root  613 Jun  7 00:55 Hello.class
-rw-r--r-- 1 root root  788 Jun  7 00:55 Hello$.class
-rw-r--r-- 1 root root  126 Jun  7 00:51 Hello.scala
```

运行程序，方法是“`scala <单例对象名> 命令行列表`”：

```
> scala Hello scala spark
Hello, scala
Hello, spark
```

小结

本附录的目的只是帮助从没接触过Scala的读者快速了解Scala的基础语法，借此帮助读者朋友们在阅读完本章内容之后，消除阅读本书的示例代码，或者查询官方API时的障碍。如果要进一步学习，大家可以阅读Scala语言的官方指导手册。

相较于其他大数据解决方案，Spark的一大特点便是擅长在单一框架内搭建一体化大数据流水线。本书以Spark 1.4为蓝本，既带读者概览Spark的各个组件，又从实际出发，给出了各种典型案例的解决思路，尤其适合初学者快速把握Spark的全貌和各组件的基本特点，从而结合实际，有针对性地发挥Spark全能一体的优势。

—— 连城（Apache Spark committer，Databricks 软件工程师）

Spark作为一个专门处理分布式大数据的有力工具，与“机器学习”碰撞出了火花。本书拿出一章重点介绍了计算广告中一个核心模型——点击率（CTR）预估的逻辑回归——在MLlib中的完整实现，为“Spark + 机器学习”提供了很好的范例。相信这本全面介绍Spark的实用宝典，会为读者提供很大帮助。

—— 胡熠（阿里巴巴B2B搜索联盟算法负责人）

以Hadoop为核心的技术，主导了过去十多年大数据技术的发展，我在小米经历了小米大数据团队的从无到有，见证了从刚开始Hadoop支持所有业务，到后来大量的业务开始使用Spark的过程，深刻体会到Spark作为后起之秀，在近几年的发展突飞猛进，大有取代Hadoop之势。一本好的技术书，既要能讲清楚技术背后的原理，又要能说明白其应用场景，《Spark最佳实践》这两方面都做到了，是学习Spark技术不可多得的好书。

—— 武泽胜（棒米科技联合创始人&CTO）

数据已经成为工业革命的重要原材料，我们无法想象一个没有数据存在的“真空”环境。可以说，我们及我们周边的事物都只是数据繁衍的一个载体；如何利用好这个原材料，加工好这个原材料，挖掘好这个原材料，理解好这个原材料，已经成为一个核心竞争力。

作为Hadoop系的重要补充，Spark更是为大数据处理虎上添翼，尤其是在更加复杂的数据迭代技术方面。作者结合多年实践著就本书，可以帮你快速进入这一“圣殿”，少走弯路，更加快速地决胜千里。

—— 肖磊（腾讯数据平台部精准推荐中心总监）

面对诸多的大数据技术，如何能够快速学习？本书的作者们试着给Spark初学者设计了一条路径：在深入讲解理论的同时，引导读者利用实际可运行的数据案例低成本地在实践中学习。“纸上得来终觉浅，绝知此事要躬行”，学习Spark技术的有效方式就是在实际的Spark环境中“玩”数据。

—— 靳志辉（腾讯社交与效果广告部质量研发中心总监）

图灵社区：iTuring.cn
热线：(010) 51095186 转 600

分类建议 计算机 / 数据分析

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-42228-6

ISBN 978-7-115-42228-6
定价：49.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks